Sistemas Informáticos Curso 2004-2005

Simulador de vuelo de helicóptero

Xavier Paolo Burgos Artizzu
Ángel Luis Diez Hernández
Ángel Iglesias Sánchez

Dirigido por:
Prof. Fernando Sáenz Pérez
Dpto. de Sistemas Informáticos y Programación

Facultad de Informática
Universidad Complutense de Madrid

INDICE

1. RESUMEN

1.1 Resumen en Castellano

Este proyecto es la continuación de uno anterior, del año 2001/2002, hecho para esta misma asignatura, en esta misma facultad, dirigido también por Fernando Saénz Pérez.

El proyecto anterior sentó las bases de simulación gráfica y física del vuelo de un helicóptero. Nosotros, al retomarlo hemos aportado mejoras y ampliaciones en todos los aspectos, así cómo ampliado sus funciones.

Los objetivos principales fijados desde el comienzo a lograr se dividían en dos grandes bloques:

1. Incorporar un modelo físico más complejo basado en el trabajo de Guillermo Martínez Sánchez "Modelación del sistema no lineal de un helicóptero". Trabajo de investigación del curso 2002/2003 del departamento A.C.Y.A de esta Universidad. Interesaba, además de incorporarlo, que se cargase desde archivo, para poder cambiarlo en tiempo de ejecución.

2. Mejorar la simulación gráfica en general. Es decir, tanto completar el escenario en el que el helicóptero se mueve (introduciendo explosiones, lagos, otros objetos...), como mejorar los ya existentes (aspecto del cielo, luces, flares, el propio terreno... También se quería ampliar el número de diales gráficos de los que se disponía.

El primer objetivo se ha cumplido en su totalidad, incorporando no sólo dicho modelo físico, si no que además se puede cargar desde fuera mediante la incorporación al proyecto del lenguaje de programación Lua. Además, se han añadido colisiones con y de todos los objetos del escenario. Y no sólo eso, si no que además al añadir proyectiles y objetos dinámicos se han añadido también sus modelos físicos.

El segundo objetivo también se ha cumplido, incluso con más de lo que se especificó originalmente. Se han añadido 8 nuevos Diales, mejorado el aspecto general del terreno y del cielo, añadido planos de agua, luces, objetos tales cómo árboles etc.

Además de estos objetivos principales también se han logrado otros no especificados al principio. A saber:

- Se ha añadido posibilidad de manejar el helicóptero mediante joystick.
- Se han añadido Sockets para poder controlar el helicóptero automáticamente desde el exterior, usando un controlador externo.
- Se ha dotado al Mundo de técnicas para que parezca Infinito.
- Se han añadido objetos dinámicos que interactúan con el helicóptero con comportamientos autónomos (aunque muy básicos.
- Se ha depurado y mejorado la eficiencia general del programa.
- Se ha añadido el lenguaje de programación Lua, que ahora se puede usar para cualquier interacción con scripts.
- Se pueden reconfigurar los Diales, tanto su posición como la existencia en pantalla de cada uno.

- Todos los controles son reconfigurables mediante un menú.
- Se ha añadido un instalador propio para facilitar su instalación en distintos ordenadores.
- Se ha añadido la posibilidad de reproducir efectos de sonido mediante FMOD (incluyendo archivos MP3.

El resultado es un simulador bastante completo y atractivo.

En un futuro, como tareas pendientes, pensamos que se podrían añadir distintos modelos de helicóptero, capacidad para jugar en red (multijugador) y dotar de mayor inteligencia a los objetos dinámicos.

1.2 English summary

This project is the continuation of one previous, of the year 2001/2002, developed for this same subject, in this same faculty, and directed also by Fernando Saénz Pérez.

The previous project sat down the bases of graphic simulation and physics of the flight of a helicopter. We have taken it over and made improvements and enlargements in all of the aspects, and also expanded its functionality.

The main objectives to achieve when we took over were divided into two large blocks:

1. To incorporate a more complex physical model based on the work of Guillermo Martínez Sánchez "Modeling of the not lineal system of a helicopter". A work of investigation of the year 2002/2003 by the department A.C.Y.A of this University. The loading of this model had to be done from an extern file, so that it would be possible to change it whenever we wanted.

2. To improve the graphics simulation in all of his aspects. In order to achieve this, it was necessary to complete the world in which the helicopter moves (introducing explosions, lakes, other objects...), as to improve the already existing graphics (the appearance of the sky, lights, flares, land...). It was also necessary to increase the number of graphic dials of the helicopter.

The first objective has been successfully completed in its totality. Not only we have incorporated the physical model, but also permitted to load it from outside by adding to the project the use of the programming language Lua. Besides, collisions have been added (with and of) all the objects of the world, and also, all the projectiles and dynamic objects have been added with its own physical models.

The second objective has also been successfully completed, even with more done than what was specified originally. 8 new dials have been added, the general appearance of the land and of the sky has been improved, and plans of water, lights, and objects such as trees have been added.

Besides these main objectives there's more work done that what was originally specified. Here's a short list:

· The possibility to handle the helicopter by Joystick

· The use of Sockets to be able to control the helicopter automatically from the outside, using an external controller.

· The use of techniques to make the world look infinite.

· Dynamic objects have been added that interact with the helicopter with autonomous behaviours (although this behaviours are very basic).

· The general efficiency of the program has been improved.

· The Lua programming language has been added, so that now it can be used for anything.

· The graphic dials can be reconfigured, by determining its positions and if they should be shown or not.

· All the controls can be recalibrated using a new window.

· An installer has been added to facilitate his installation in different computers.

· The possibility of playing sound effects has been added by the use of FMOD (including MP3 files).

The result is an attractive and quite complete flight simulator.

In a future, as pending tasks, we think that it could be good to add different models of helicopter, multiplayer options and to endow of greater intelligence to the dynamic objects.

2. DESCRIPCIÓN DEL PROYECTO

2.1 DESCRIPCIÓN

Este proyecto ha sido llevado a cabo en Visual C++, siendo la continuación de un proyecto anterior de esta misma facultad y dirigido por el mismo tutor en el año 2001/2002.

Su objetivo es simular gráficamente, con ayuda de las librerías gráficas de DirectX,
el vuelo de un helicóptero en un escenario real. El usuario es el que maneja y dirige el helicóptero por medio del teclado, ratón e incluso de un joystick (sí se desea.

La simulación es tanto física como gráfica. Por física se entiende que simula el vuelo del helicóptero integrando las variables y ecuaciones físicas que intervienen en su movimiento, como su interacción física con el resto del escenario (colisiones y demás. Esto es necesario para dotar al simulador de realismo.

Por gráfica se entiende "dibujar" gráficamente en el ordenador un helicóptero volando así como todo el escenario en el que este se encuentra y con el que interacciona. Este escenario se ha dotado de técnicas para que parezca Infinito. Se simulan también los diales más comúnmente presentes en la cabina de todo helicóptero.

Además se ha dotado a este simulador de capacidad para ser controlado desde el exterior mediante un controlador. Como ejemplo se han implementado dos controladores básicos, uno en C y otro en Simulink.

Para conseguir dichos objetivos se ha trabajado sobre distintos aspectos y utilizado distintas técnicas y lenguajes de programación.
Esta documentación complementa al ejecutable y su código explicando una a una las tareas llevadas a cabo, las técnicas utilizadas y cómo se han implementado.

2.2 HISTORIAL DE VERSIONES

14 / 10 / 2004 Versión 1.0

Versión inicial del proyecto, realizada como proyecto de la asignatura de Sistemas Informáticos durante el curso 2000/2001 por Andrés Ruiz Flores y Jesús de Santos García.

28 / 10 / 2004 Versión 1.1

Se añade la representación de un vector de fuerza para comprobar el correcto funcionamiento del modelo físico.

4 /11 / 2004 Versión 1.2

Inclusión de otros modelos gráficos además del helicóptero que se controla, aunque con problemas.

18 / 11 / 2004 Versión 1.3

Solucionado los problemas de representación de otros objetos.

16 / 12 / 2004 Versión 1.4

Se incluyen explosiones; acercamiento en la definición de escenarios a través de fichero. Se añade la librería FMOD para reproducir efectos de sonido.

3 / 3 / 2005 Versión 1.5

Inclusión de los indicadores de orientación (brújula), de velocidad, así como el agua. Se desecha la idea de definir escenarios por fichero, sustituyendo el cargador defectuoso por las librerías de lua.

10 / 3 / 2005 Versión 1.6

Se soluciona el problema de rebote de la cámara, además se incluyen dos cámaras nuevas. Se refina el orden de pintado de elementos en la pantalla debido a errores observados. Se introduce el control mediante joystick de dos ejes.

20 / 5 / 2005 Versión 1.7

Mejora del control por joystick para adaptarlo a un joystick con dos palancas, HUD reconfigurable mediante fichero. Creación de un instalador, pero con problemas para instalar las librerías de DirectX. Inclusión en el HUD de un radar tridimensional y un horizonte artificial. Se introduce una pantalla para informar del proceso de carga antes de empezar la ejecución. Introducción de luces. Introducción de un tanque con inteligencia artificial y un modelo físico bastante simple.

29 / 5 / 2005 Versión 1.8

Inclusión del mundo infinito; se añade al HUD un altímetro, un indicador de la posición en el mundo y un indicador de velocidades y aceleraciones lineales y angulares. Se intenta exportar archivos de Maya 6.5 a ficheros.x sin éxito.

1 / 6 / 2005 Versión 1.8.1

Se modifica el indicador de potencia para que ocupe menos, y se incluye un indicador de combustible. Se incluye también comunicación mediante sockets, para probarlo se crean controles rudimentarios tanto en C++ como en Matlab. Posibilidad de exportar modelos tridimensionales sin texturar desde Maya a .x mediante el programa DeepExploration; problemas para pasar modelos con texturas.

15 / 6 / 2005 Versión 1.9

Incluida la reproducción de video, cambio del cursor en pantalla y de un icono para la ventana y el fichero ejecutable. Creación del menú de ventana, configuración durante ejecución de los controles. Se añade la posibilidad de reproducir archivos MP3 mediante FMOD.

24 / 6 / 2005 Versión 1.9.5

Se incluye la selección de indicadores visibles en el menú, se mejora el efecto de la cámara al introducirse en el agua.

28 / 6 / 2005 Versión 2.0

Se incluye un instalador usando el sistema de Scripts NSIS; además se mejoran el indicador de velocidades y aceleraciones y el radar tridimensional,

2.3 MANUAL DE USUARIO

Requisitos Mínimos

Windows Me, 2000, XP
DirectX v8.0
Pentium III 750 HMS
128 Nov de Ram
Tarjeta aceleradora de segunda generación (GeForce)

Requisitos Recomendados

Windows XP
DirectX v8.1
Pentium IV 1.6 Ghz
256 Nov de Ram
Tarjeta aceleradora de última generación

Instalación

El programa dispone de instalador propio. Basta ejecutar este e instalar en el directorio que se prefiera.

Ejecución

El ejecutable del programa es *Simulador.exe*
Nada más empezar muestra un video de introducción. Si se quiere saltar, pulsar INTRO. Después inicia la carga en el sistema del programa. Una vez finalizada pulsar la tecla asociada al Disparo, que por defecto es la tecla control a no ser que se haya redefinido antes.
Una vez lanzado, las teclas disponibles son las siguientes.

- **F1:** Activamos el modo gráfico en ventana.
- **F2:** Activamos el modo gráfico en pantalla completa.
- **F3:** Abrimos el reconfigurador de teclado y joystick.
- **F5:** Se activa la cámara de seguimiento.
- **F6:** Se activa la cámara libre.
- **F7:** Se activa la cámara del interior del helicóptero.
- **F8:** Se activa la cámara trasera.
- **Cursores:** Con las teclas de los cursores arriba y abajo podemos alejar / acercar la cámara del punto de vista del helicóptero. O mover esta hacia delante o atrás cuándo está en modo libre.

• **Ratón:** Con el ratón podemos mover la cámara para cambiar el punto de vista del helicóptero. Y si se está en modo cámara libre, con el botón derecho se sube la cámara y con el izquierdo se baja.

• **W / S:** Con estas teclas aceleramos / deceleramos el motor del helicóptero. Está aceleración se traduce directamente en la velocidad de giro del rotor principal, que es fuerza principal que mueve el helicóptero.

• **Q / A / O / P:** Con estas teclas movemos hacia arriba, abajo, izquierda y derecha la palanca del cíclico. La situación de la palanca del cíclico se representa gráficamente mediante un circulo en la esquina inferior derecha. Cuando soltamos alguna de estas teclas, la palanca vuelve automáticamente a su sitio.

• **Z / X:** Con estas teclas aceleramos/deceleramos la velocidad del rotor trasero. Al desfasarse este rotor con respecto al principal, se producen fuerzas laterales que hacen que el helicóptero gire.

• **I:** Se esconde el texto informativo sobre la ejecución que aparece por pantalla.

• **E:** Se muestra la Malla que forma el terreno.

• **1 / 0:** Encendemos y apagamos el motor directamente con máxima potencia.

• **Control derecho:** Disparamos un proyectil de tipo misil.

• **Control izquierdo:** Disparamos un proyectil de tipo rayo cósmico.

• ***:** Encendemos / apagamos la luz ambiental.

• **/:** Encendemos / apagamos la luz solar.

• **Escape:** Salimos del programa (antes se nos pregunta sí estamos seguros de querer salir.

• **Joystick:** Se puede también usar un joystick para manejar el objeto de la simulación. Aquí se recomiendo reconfigurar los botones del joystick ya que como predeterminado está una configuración para bastantes botones:

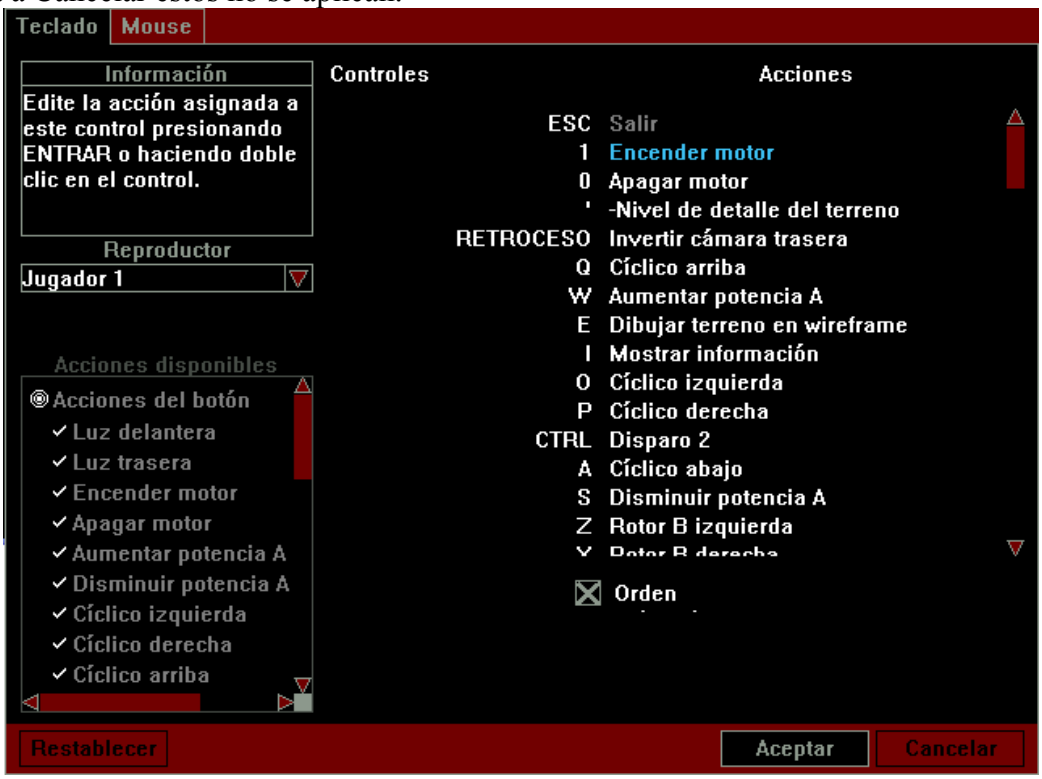
Botón 0: Encender motor
Botón 1: Apagar motor
Botón 2: Cíclico arriba
Botón 3: Disparo 1
Botón 4: Luz trasera
Botón 5: Disparo 2
Botón 6: Luz delantera
Botón 7: Configurar controles
Botón 13: Cíclico derecha
Botón 14: Cíclico abajo
Botón 15: Cíclico izquierda

Eje Z: Rotor trasero
Rotación Z: Rotor principal
Eje X: Eje Y del cíclico
Eje Y: Eje X del cíclico

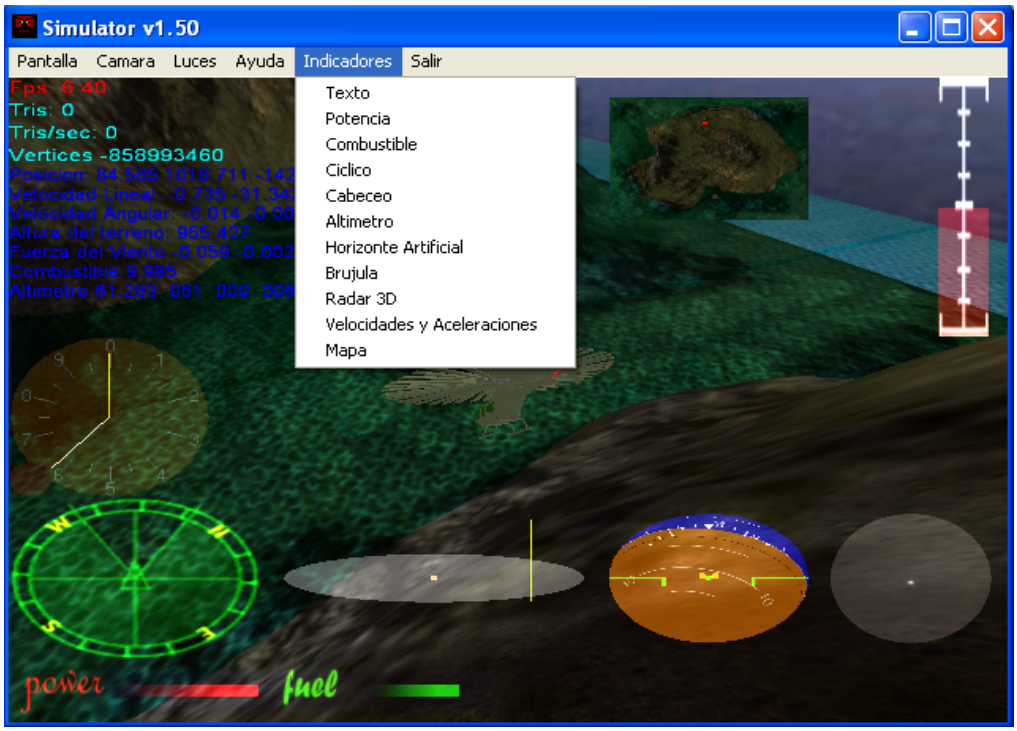
Nótese que estos son los controles por defecto, pero siempre que se quiera se pueden reconfigurar usando el reconfigurador de teclas disponible pulsando F3, o desde el menú Ayuda. Este reconfigurador es de muy sencillo de usar. Tiene 3 pestañas (cuándo no hay un Joystick conectado sólo tiene 2), Teclado/Ratón/Joystick. Al pinchar sobre una de ellas podemos redefinir los controles de dicho dispositivo. Para ello, hay que pinchar sobre la lista de la derecha (Acciones) en la tecla o botón que se quiera redefinir y o bien doble cliquear o bien pulsar Intro, y a continuación seleccionar de la lista de la izquierda (Acciones disponibles) su nueva acción asociada.

El comboBox reproductor está orientado a añadir varios jugadores en futuras ampliaciones. La casilla Orden sirve para ordenar o no las acciones asignadas.

Una vez finalizado, si se le da a Aceptar se aplican los cambios mientras que si se le da a Cancelar estos no se aplican.



Además las funciones relativas a modificar las luces, el modo de pantalla, el tipo de cámara y el reconfigurador son también accesibles a través del menú de la ventana, haciendo clic con el ratón en ellos (para esto se debe estar en modo ventana. Además, mediante el menú Indicadores se puede seleccionar que diales se muestran y cuáles no.



Archivos de configuración

Archivos de la carpeta scripts del directorio que contiene el ejecutable del programa. Estos archivos los puede modificar el usuario para configurar la aplicación sin necesidad de recompilar el programa.

Archivos obligatorios:

Control.Lua Archivo de configuración de los sockets.

Escena.Lua Configuración de objetos del paisaje, como el sol, el efecto de estar por debajo del agua, etc.

Helicoptero.Lua Configuración del objeto que vamos a manejar en la simulación. Aquí se especifica el modelo 3D, el modelo físico, el combustible, el consumo, la posición inicial, etc.

Modelo.Lua Modelo físico del helicóptero que había inicialmente en el proyecto. Especifica como calcular las fuerzas, momentos y derivadas del helicóptero.

Modelo1.Lua Modelo físico explicado en el trabajo "Modelación del sistema no lineal de un helicóptero".

ObjetosDin.Lua Configuración de los objetos dinámicos que queremos que estén en la escena.

ObjetosMundo.Lua Configuración de los objetos estáticos de la escena.

Proyectiles.Lua Configuración de las ecuaciones físicas que rigen los proyectiles, así como las texturas y el modelo 3D utilizadas para representar el proyectil.

Terreno.Lua Configuración del terreno, el plano de agua y el cielo de la escena.

Archivos opcionales:

ModeloDin.Lua Modelo físico muy simple para un tanque que actúa con inteligencia artificial.

ModeloTank.Lua Modelo físico muy simple de un tanque sin inteligencia artificial.

Tanque.Lua Configuración de un tanque.

3. EFECTOS ESPECIALES

En esta sección se tratan diversos efectos especiales incorporados al simulador tales como partículas, explosiones, proyectiles, luces y agua. Se explica cómo se han implementado y las características de cada uno.

3.1 SISTEMA DE PARTÍCULAS

Uno de los efectos especiales realizados son las partículas. A continuación se explica el método utilizado para crearlas, actualizarlas y dibujarlas.

3.1.1 ¿Qué es un sistema de partículas?

En la vida real, una partícula es un pequeño fragmento de algo. Por ejemplo, cuando golpeas un ladrillo con un martillo, el ladrillo se rompe en fragmentos que van en todas las direcciones. Una gota de agua, un copo de nieve, una salpicadura de sangre o una chispa se pueden considerar también partículas.

Programando, una partícula es una textura asociada a un cuadrado en dos dimensiones. Estos cuadrados normalmente son pequeños y se dibujan parcialmente transparentes.

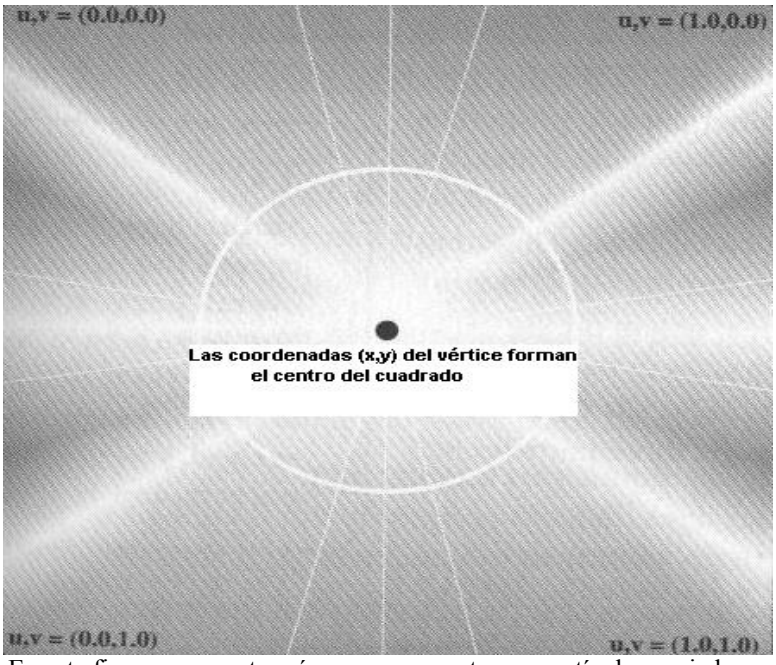


Figura 3.1 En esta figura se muestra cómo se representa una partícula asociada a una textura.

Modelando la física de cada partícula podemos crear efectos especiales complejos. Por ejemplo podemos crear una tormenta de nieve con un array de centenas de partículas. Cada frame recorremos este array ajustando cada partícula a su nueva posición. Si dibujamos cada partícula con una textura que represente un copo de nieve tendremos un efecto de tormenta de nieve convincente. Realmente estamos haciendo lo que pasa en la naturaleza, de ahí el hecho de que podamos crear efectos que se asemejan a la vida real.

Un sistema de partículas consta de tres cosas:

1. Un array de partículas. Las partículas serán objetos.
2. Una función de actualización. Actualizará todas las partículas del array.
3. Una función de dibujo. Dibujará las partículas con las propiedades adecuadas.

3.1.2 Propiedades de las partículas

La idea general es que cada partícula se crea en una posición determinada del mundo. Mientras el sistema las mueve por el mundo, ellas viven su corta y feliz vida. En algún momento determinado las partículas morirán. Así que ya tenemos la primera propiedad: tiempo de vida.

La función de actualización tiene en cuenta cuanto tiempo lleva viviendo la partícula (en segundos), si dicho tiempo excede el tiempo de vida, la partícula muere.

Las propiedades principales de las partículas son:

1. Posición
2. Tiempo de vida
3. Velocidad
4. Color
5. Tamaño

3.1.3 Propiedades del sistema de partículas

Ahora que sabemos qué tiene una partícula, lo siguiente es saber que propiedades tiene el sistema en sí para todas las partículas.

- Gravedad
- Textura
- Radio de emisión. Este parámetro controla la velocidad a la que el sistema crea nuevas partículas.

Ahora bien, para crear nuevas partículas necesitamos darles ciertos valores (color, tiempo de vida…) Una opción es darle al sistema un valor mínimo y uno máximo y que el sistema elija un valor al azar en dicho rango. Así crearemos partículas distintas pero siempre con un cierto grado de libertad.

- Mínimo, máximo tiempo de vida
- Mínimo, máximo color
- Mínima, máxima velocidad
- Mínimo, máximo tamaño

3.1.4 Array de reciclaje

Para almacenar las partículas hemos creado una clase especial llamada CRecyclingArray.

Esta clase funciona como la clase std::vector, pero con una diferencia importante: no usa memoria dinámica. En vez de esto, le damos un valor máximo de objetos que va a poder contener. A efectos de uso, podemos añadir y borrar objetos del array. CrecyclingArray no usa memoria dinámica, usa flags booleanos que le dicen qué lugares están libres en el array. Cuando creamos un objeto usando esta clase, el array

coge el primer objeto marcado como libre y nos lo da. Cuando borramos un objeto del array lo que hace es marcarlo como libre. No hay asignación de memoria dinámica, simplemente reciclamos la memoria mientras borramos y creamos nuevos objetos.

¿Porqué no usar new y delete? Hay dos motivos. El primero es que la asignación de memoria es lenta. Durante la vida de un sistema de partículas miles de partículas son creadas y destruidas. Podemos perder mucho tiempo y no nos conviene. Lo segundo es que asignar muchos trozos fragmenta la memoria.

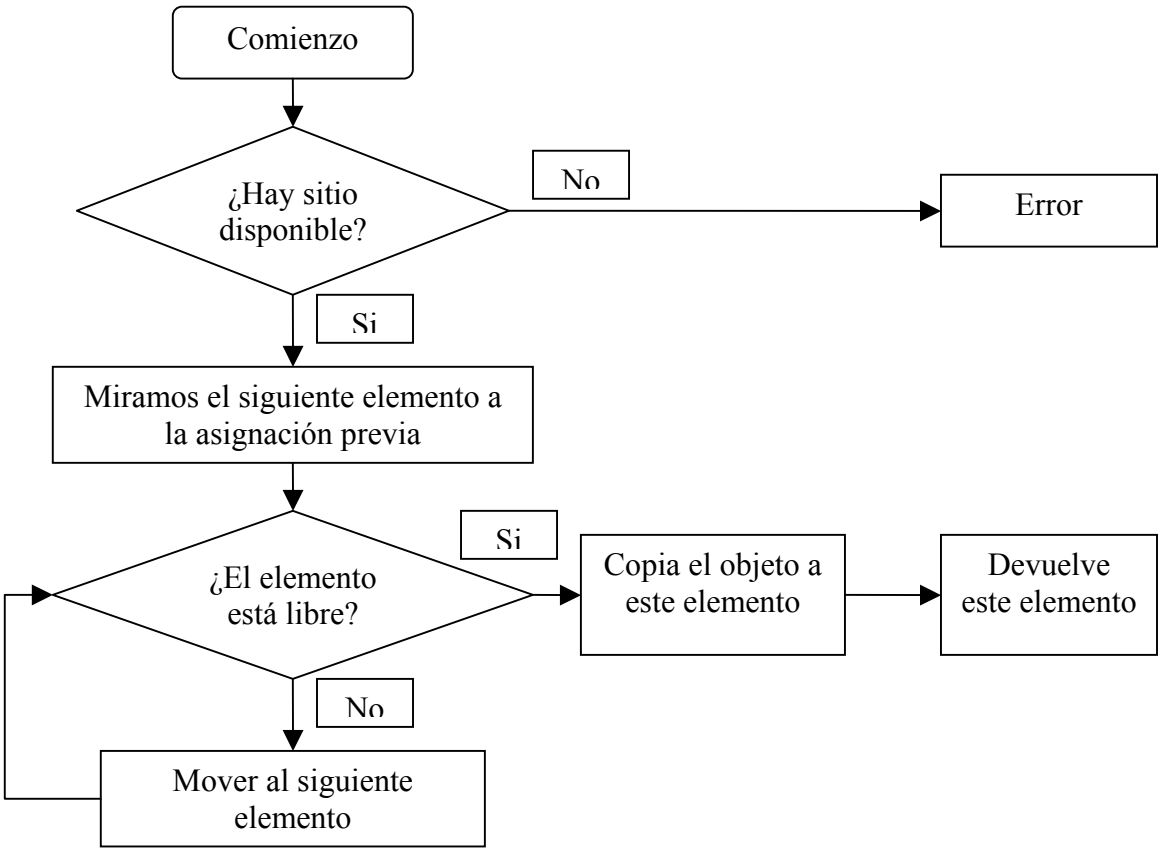


Figura 3.2 Lógica del array de recicleje para la creación de objetos.

En la figura 3.3 se puede ver un ejemplo de cómo afecta la fragmentación de la memoria:

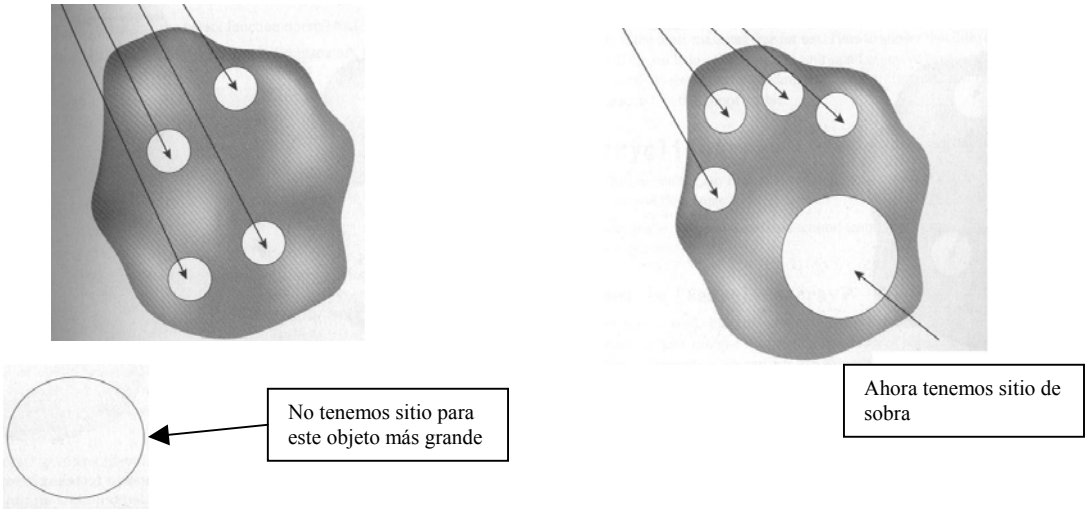


Figura 3.3 Ala izquierda representación de la memoria fragmentada, a la derecha memoria no fragmentada.

3.1.5 Render states

D3DRS_POINTSPRITEENABLE. Estableciendo este estado a cierto le decimos a Direct3D que cuando le enviamos puntos al sistema, Direct3D debe asignarle automáticamente coordenadas de textura de tal forma que toda la textura se aplique a cada punto. Es como decirle que cada punto que le mandas es el centro de la textura.

D3DRS_POINTSCALEENABLE. Este valor a cierto le indica a Direct3D que debe escalar el tamaño de las partículas según la distancia a la cámara.

D3DRS_POINTSIZE_MIN y D3DRS_POINTSIZE_MAX indicant el tamaño mínimo y máximo de un punto. Los valores que no estén en ese rango se ajustarán al mínimo o al máximo dependiendo de qué límite han sobrepasado.

D3DRS_POINTSCALE_A, D3DRS_POINTSCALE_B y D3DRS_POINTSCALE_C indican cómo se ha de calcular el tamaño según la distancia a la cámara. La ecuación 3.1 muestra el cálculo del tamaño final:

$$S_s = V_h * S_i * \sqrt{\frac{1}{A + B * D_e + C * D_e^2}}$$

Ecuación 3.1

Ss: Tamaño final
Vh: Altura del viewport
Si: Tamaño de la partícula especificado en el vertex data.
A: D3DRS_POINTSCALE_A
B: D3DRS_POINTSCALE_B
C: D3DRS_POINTSCALE_C
De: Distancia entre la cámara y la partícula.

3.2 MEJORA DEL SISTEMA DE PARTÍCULAS

El sistema de partículas realizado hasta el momento da buenos resultados, pero es poco flexible, es decir, sus propiedades son fijas. Vamos a ver cómo mejorar dicho sistema.

3.2.1 Añadiendo flexibilidad

El siguiente paso en nuestro sistema de partículas es hacerlo más flexible, es decir, que podamos cambiar las propiedades dinámicamente, que no tengan siempre los mismos valores. Por ejemplo, si nos fijamos en el color nos gustaría pasar desde un color inicial a un color final pasando por diversos colores intermedios dependiendo del tiempo. Esto hará más flexible el sistema. Si miramos todas las propiedades, no sólo el color, tendremos un sistema mucho mejor.

3.2.2 Eventos

El concepto de evento sirve para contener todos los cambios que se pueden hacer, de color, tamaño, lo que sea.

El concepto de polimorfismo nos sirve de gran ayuda. Cada evento tendrá su propia función de actualización, en nuestro caso se llama DoItToIt(). Dicho método recibirá una partícula y cambiará sus propiedades dependiendo del evento.

Ahora el sistema tendrá un array de eventos.

Un evento es esencialmente un cambio de alguna propiedad de una partícula dependiendo del tiempo.

3.2.3 Secuencias de eventos

Si imaginamos un sistema de partículas de fuego, por ejemplo, que cree no sólo brasas sino también humo podemos ver que necesitamos más de una secuencia de eventos. Una secuencia se encargaría de las brasas, evento de color, evento de tamaño, etc… Y otra secuencia se encargaría del humo.

Para atacar esto necesitamos un sistema que sea capaz de controlar varios vectores de eventos. Cuando se crea una nueva partícula, el sistema debe decidir que secuencia de eventos va a seguir. Podemos asignar una probabilidad para usar una secuencia u otra.

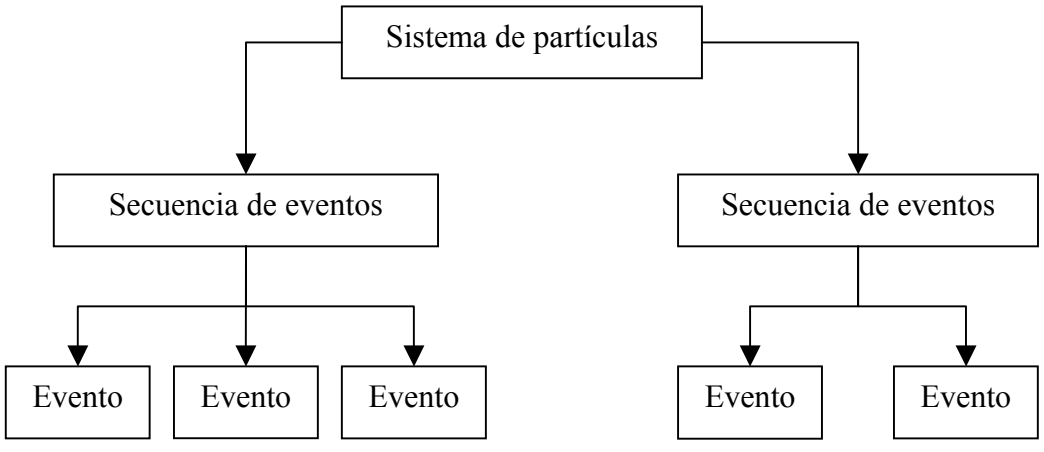


Figura 3.4 Resúmen de la arquitectura del sistema.

3.2.4 Scripts de sistemas de partículas

Para hacer el sistema de partículas independiente del programa necesitamos un sistema para leer desde archivo el sistema de partículas y crearlo según dicho archivo. Para leer el archivo implementamos la clase CParticleEmitterTokenizer. Esta clase sigue el esquema de estados de la figura 3.5.

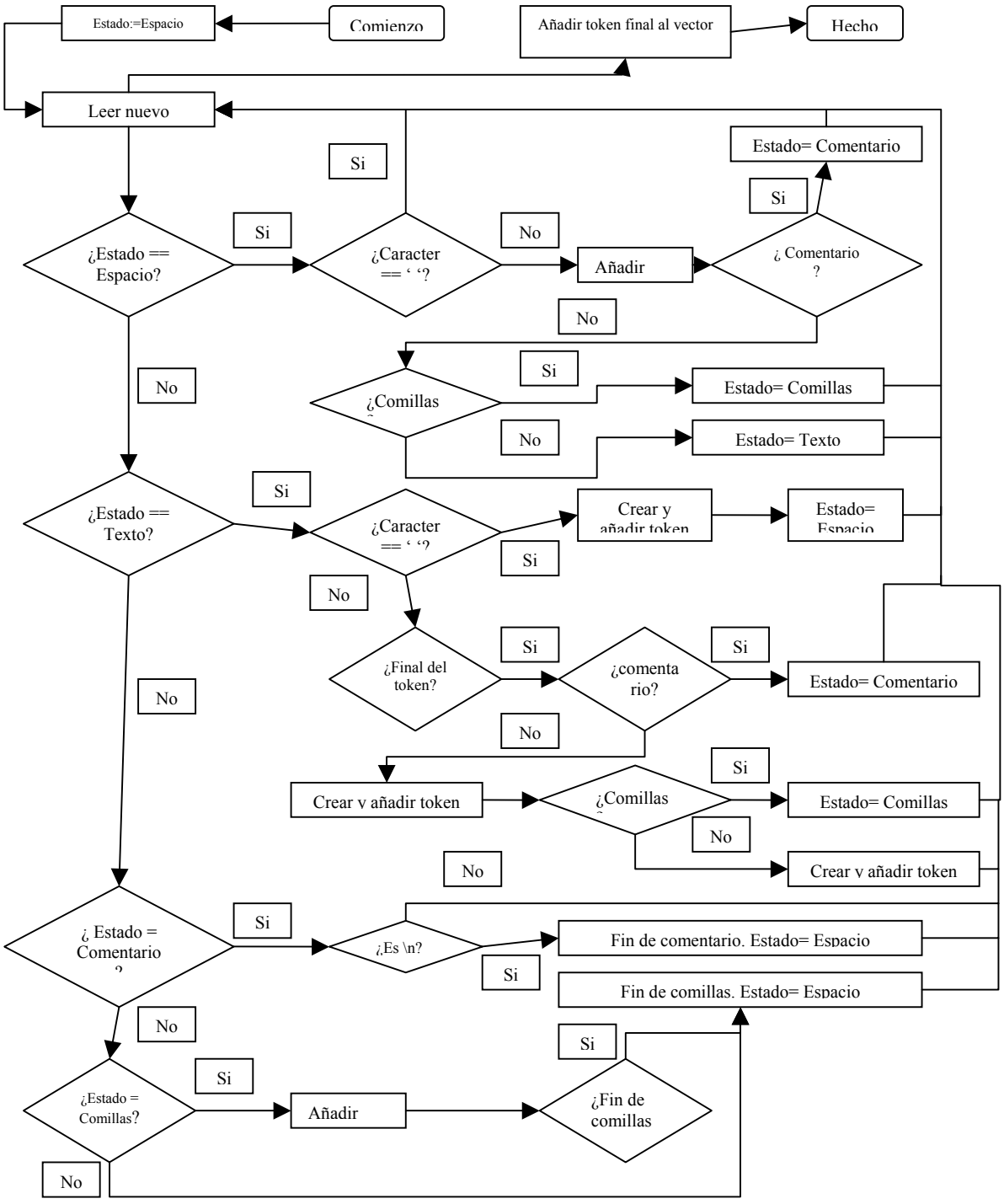


Figura 3.5 Esquema de la lectura de un script de un sistema de partículas.

3.2.5 Gramática del lenguaje

```
RealNumber ::= --<float>--
Number ::= <RealNumber> | Random(<RealNumber>,<RealNumber>)
Vector ::= XYZ(<Number>,<Number>,<Number>)
Name ::= "{--< char >--}"
AlphaBlendMode ::= D3DBLEND_ZERO |
        D3DBLEND_ONE |
        D3DBLEND_SRCCOLOR |
        D3DBLEND_INVSRCCOLOR |
        D3DBLEND_SRCALPHA |
        D3DBLEND_INVSRCALPHA |
        D3DBLEND_DESTALPHA |
        D3DBLEND_INVDESTALPHA |
        D3DBLEND_DESTCOLOR |
        D3DBLEND_INVDESTCOLOR |
        D3DBLEND_SRCALPHASAT
Color ::= RGBA(<Number>,<Number>,<Number>,<Number>)
VersionNumber ::= <RealNumber>

ParticleSystem ::= PartycleSystem <VersionNumber> <Name>
        "{" {<SysProperty>} {<EventSequence>} "}"
SysProperty ::= POSITION  = <Vector>
EventSequence ::= <Name> "{" {<SeqProperty>} {<Event>} "}"
SeqProperty ::= GRAVITY = <Vector> |
                <SingleSeqProperty> = <Number> |
                <BlendProperty>=<AlphaBlendMode> |
                TEXTURE = <Name>
BlendProperty ::= SOURCEBLENDMODE |
                DESTBLENDMODE
SingleSeqProperty ::= NUMPARTICLES |
        EMITRATE |
        EMITRADIUS |
        LIFETIME
Event ::= "initial" <EventProperty> |
        "fade so at" <Number> <EventProperty> |
        "fade so final" <EventProperty>
EventProperty ::= COLOR = <Color> |
        SIZE = <Number> |
        VELOCITY = <Vector>
```

3.3 EXPLOSIONES

Un efecto especial interesante son las explosiones, para representarlas se pueden usar diversas técnicas. Las explosiones más sencillas se pueden hacer con una animación creada dibujando distintos cuadrados con diferentes transparencias y texturas de explosiones. Lo primero que hay que hacer es crear texturas adecuadas para las explosiones. En nuestro caso hemos usado un CD de Pyromania. Lo que hicimos fue convertir las imágenes del CD de Pyromania en texturas.

3.3.1 Clases

Una vez que tenemos las imágenes necesitamos código para leerlas y cargarlas en memoria. Para realizarlo implementamos dos clases:

La clase CSprite que gestiona un frame de la animación, y la clase CAnimSequence que contiene una secuencia de frames para la animación.

La clase CAnimFrame consiste en un manejador de texturas y un valor que gestiona el tiempo. El valor que gestiona el tiempo nos indica cuánto debe permanecer visible un determinado frame. En esta parte nos hace falta utilizar un método de retardo, por lo que hemos implementado una clase adicional para gestionar el tiempo, CTimer.

La parte principal de la clase CAnimSequence consiste en un vector de frames de la animación y un buffer con los vértices de los cuadrados que se van a dibujar. Para usarla creamos una instancia y le añadimos los frames y los tiempos apropiados llamando al método AddFrame().

Una cosa importante es tener separados la animación de la posición y el tiempo. De esta manera podemos tener la misma animación ejecutándose en distintos momentos y distintos sitios durante el juego. Así la clase CSprite tiene tres atributos importantes, el tiempo, la posición y la secuencia de animación completa.

3.3.2 Billboarding

En este punto llegamos a un problema, las imágenes que tenemos son en 2D, pero tenemos que dibujarlas en 3D. Nuestras imágenes 2D las colocamos en cuadrados como texturas. Para crear la ilusión de que estas imágenes son explosiones en 3D necesitamos asegurar de alguna forma que estas imágenes miren siempre a la cámara. Si la cámara viera las imágenes desde un ángulo colocar bien las imágenes veríamos como si las explosiones fueran una especie de cartel.

Esta técnica se conoce como *billboarding*(cartelera), y es una herramienta importante en el dibujo de escenas 3D de alta calidad. Billboarding toma su nombre del hecho de que realmente creamos un cartel con un dibujo en él. Este cartel siempre está mirando a la cámara.

La técnica de billboarding se puede usar en muchas más cosas, por ejemplo, puedes tomar el dibujo de un árbol y situarlo en el cartel. Por supuesto que si la cámara se acerca mucho al cartel, nos daremos cuenta de que realmente no es un modelo 3D. Sin embargo, si el usuario no va a llegar a estar tan cerca del cartel puedes usarlo y ahorrarte dibujar unos 2000 triángulos más.

Para dibujar el cartel mirando a la cámara necesitamos obtener primero la matriz de vista(la posición y la orientación de la cámara) y crear su traspuesta. Ahora cogemos el bloque 3x3 de la esquina superior izquierda de la matriz traspuesta y usarla como matriz de rotación. Luego usamos dicho bloque en combinación con la posición, rotación y escala de la animación. Ahora ya da igual dónde esté situada la cámara porque los carteles (cuadrados con textura) siempre van a estar mirando a la cámara.

3.3.3 Encadenamiento de explosiones

Para hacer más realista la explosión implementamos una secuencia de explosiones en diferentes posiciones en torno a un punto fijo. Lo único que hacemos es usar varias secuencias de explosiones con tamaños de textura distintos, situadas en distintas posiciones y con distinta duración.

Una mejora más consiste en usar un sistema de partículas asociado a la explosión. En nuestro caso hemos creado un sistema con dos secuencias de eventos similares entre sí, las dos usan la misma textura, mismo radio de emisión y misma tasa de emisión. Pero una es más duradera, con partículas más grandes y más lenta que la otra.

Todo esto se lee desde un archivo de texto que se puede cambiar fácilmente sin necesidad de compilar.

3.3.4 Onda expansiva

Para hacer las explosiones más llamativas añadimos una onda expansiva. Lo primero de todo es escoger una textura adecuada, nosotros usamos una textura que va desde el blanco hasta el rojo y finalmente negro.

Ahora que tenemos la textura tenemos que crear una malla sobre la que dibujarla. Nuestro objetivo es crear un anillo con triángulos, y escalar progresivamente dicho anillo para hacer la onda.

Lo que necesitamos son las coordenadas de varios puntos interiores del anillo y de varios puntos exteriores del anillo. Nos centraremos en las coordenadas ‘x’ y ‘z’. Así tendremos una onda con profundidad y ancho pero no altura.

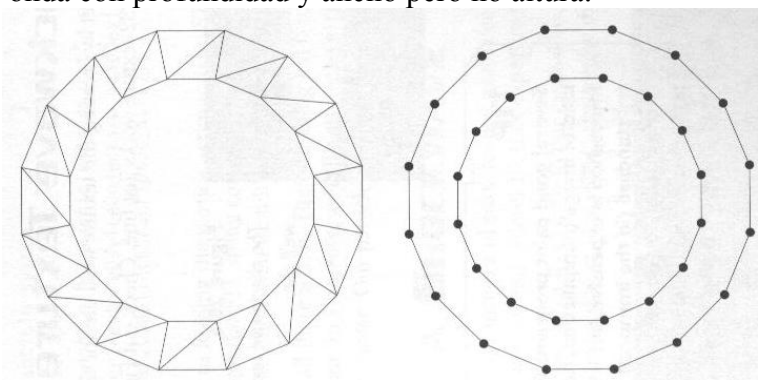


Figura 3.6 Representación de un anillo con una malla de triángulos.

Usamos la fórmula 3.2 para calcular ‘x’ y ‘z’ dadas una distancia d y un ángulo theta:

$$x = d \cdot (\sin(\theta))$$
$$z = d \cdot (\cos(\theta))$$

Ecuación 3.2

De esta forma podemos calcular las coordenadas de todos los vértices del anillo.

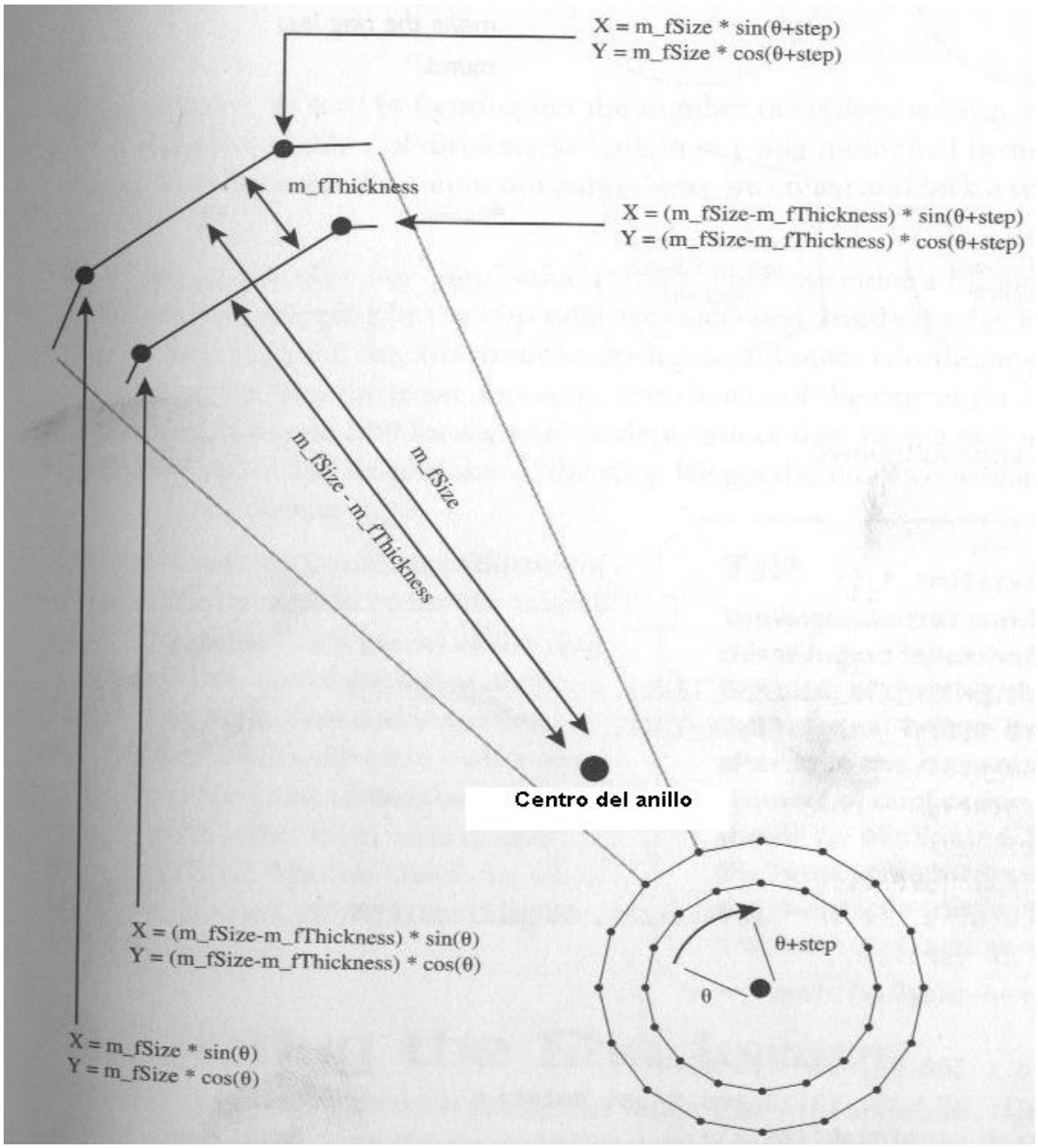


Figura 3.7 En el dibujo se muestra cómo calcular todos los vértices del anillo.

Esta forma de calcular las coordenadas tiene una ventaja, podemos aumentar o reducir el número de triángulos que forman el anillo variando el número de ángulos que usamos para calcularlos. Si queremos un anillo muy detallado podemos crear un punto por cada ángulo desde 0 hasta 360(incrementando en un grado). Para crear un anillo menos detallado podemos coger un incremento de 10, 20 o incluso 50 grados.

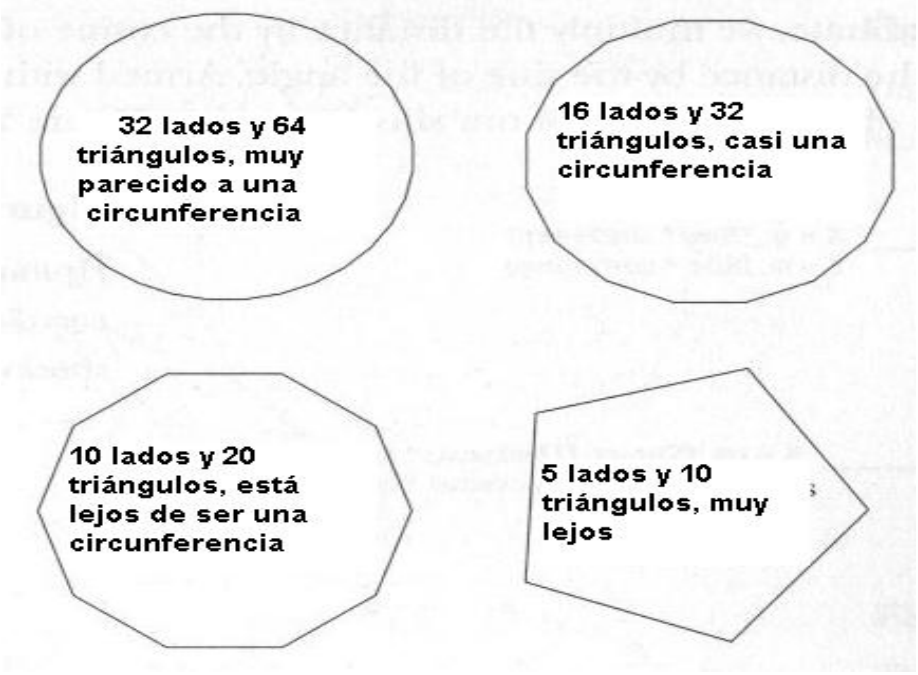


Figura 3.7 Esta figura muestra cuatro anillos creados con diferentes ángulos.

3.4 PROYECTILES

Para hacer más entretenido el simulador se incorporó un sistema para permitir tanto disparar a los objetos de la escena como que los objetos puedan dispararte a ti también.

3.4.1 Clase básica

Las clases están diseñadas usando polimorfismo para poder elegir varios tipos de armas. La clase base es CGun que tiene métodos virtuales Render(), Update(), CanFire() y Fire(). Estos cuatro métodos son el corazón de la clase. Todos los tipos de armas deben implementar estos métodos. Esta clase también almacena datos comunes a todas las armas como la posición y el tiempo.

3.4.2 Balas y array de balas

En nuestras armas una cosa importante es que se pueden ver los proyectiles. Por ello necesitamos un array para almacenar las posiciones de todas las balas disparadas hasta el momento.

Para hacerlo más modular hemos creado una clase que gestiona el array de proyectiles, que es usada por la clase principal del arma.

Esta clase que gestiona el array tiene en este caso una textura del disparo y/o un modelo 3D del proyectil. De esta manera tenemos dos tipos de proyectiles.

Para dibujar el disparo con la textura usamos el método de billboarding y utilizamos la clase que gestionaba la animación de la explosión. Para actualizar las balas necesitamos recorrer el array para ver qué modificaciones hay que hacer de posición, si el proyectil ha colisionado o no y si ha expirado su tiempo de vida(para que no sean eternas).

Para dibujar el proyectil con el modelo 3D dibujamos la malla con la orientación adecuada.

3.4.3 Disparando

Para crear un proyectil necesitamos saber su posición y su velocidad. Una vez que tenemos esto añadimos un nuevo proyectil al array de balas.

Una cosa importante es que hemos puesto un temporizador para que tenga que pasar un tiempo desde la última vez que se disparó el arma, de ahí el método CanFire(), y si podemos disparar añadimos la bala.

A la hora de crear la bala hay que indicar si queremos que se dibuje como una textura o como un modelo 3D.

3.4.4 Actualización de los proyectiles

Para hacer la clase más independiente del programa hemos usado el sistema de scripts de LUA. Con LUA leemos distintas propiedades de los proyectiles:

- Velocidad inicial de los proyectiles.
- Radio de colisión de los proyectiles.
- Modelo 3D de los proyectiles.
- Textura de los proyectiles.

En la función Update() usamos también LUA para que actualice los proyectiles según las ecuaciones físicas indicadas en el archivo de texto. Así si queremos que nuestros proyectiles no sean afectados por la gravedad, por ejemplo, simplemente cambiando el archivo de texto modificaríamos el juego, todo ello sin necesidad de recompilar el programa.

La física utilizada para los proyectiles está recogida en la sección "Física de la implementación".

3.5 ILUMINACIÓN

Para añadir más realismo a una escena, podemos añadir luz. La luz ayuda a resaltar la forma sólida y el volumen de los objetos. Cuando usamos iluminación, ya no tenemos que especificar el color de los vértices para dar realismo, sino que Direct3D pasa cada vértice por el motor de iluminación y calcula un color para el vértice basado en luces, materiales y la orientación de la superficie respecto a las luces. Calcular los colores de esta forma da como resultado una escena más natural.

3.5.1 Componentes de la luz

En el modelo de iluminación de Direct3D, la luz emitida por una fuente de luz consiste de tres componentes, o tres tipos de luz:

- Luz de Ambiente (o Entorno): Este tipo de luz modela la luz que es reflejada de otras superficies y de forma igual para toda la escena. Por ejemplo, partes de un objeto pueden ser iluminadas, aunque no tengan una luz apuntando directamente hacia este. Estas partes han sido iluminadas porque la luz se ha reflejado de otras superficies. La luz de ambiente es una forma rápida de agregar un tono de luz a toda la escena.
- Luz Difusa: Este tipo de luz viaja en una dirección en particular. Cuando golpea una superficie, la refleja igualmente en todas las direcciones. Como la luz difusa refleja la luz igualmente en todas las direcciones, la luz alcanzará el ojo sin importar de dónde o cómo se esté mirando, así que el observador no se toma en

cuenta. Así la ecuación del modelo de luz difusa solo necesita considerar la dirección de la luz y la superficie. Este tipo de es la luz general que emite una superficie.

- Luz Reflejada (Especular o Brillos): Este tipo de luz viaja en una dirección en particular. Cuando golpea una superficie, se refleja en una dirección, causando un brillo que sólo puede ser visto desde ciertos ángulos. Como la luz se refleja en una dirección en particular, claramente, el punto de vista debe tenerse en cuenta además de la dirección de la luz y la superficie en la ecuación que modela la luz especular. La luz especular es usada para modelar la luz que crea brillos en objetos, resaltes, como sucede cuando una luz toca un objeto metálico, o el sol golpea el agua. Este tipo de luz requiere muchos más cálculos que otros tipos de luz, y por esto se puede poner y quitar. Es más, por defecto no se utiliza; para hacerlo debemos establecer el parámetro D3DRS_SPECULARENABLE.

```
lpd3ddevice8->SetRenderState(D3DRS_SPECULARENABLE, true);
```

Cada tipo de luz está representada por una estructura de color D3DCOLORVALUE o D3DXCOLOR, la cual describe el color de la luz. Aquí hay algunos ejemplos de diferentes colores de luz:

```
D3DXCOLOR rojoAmbiente(1.0f, 0.0f, 0.0f, 1.0f);
D3DXCOLOR azulDifusa(0.0f, 0.0f, 1.0f, 1.0f);
D3DXCOLOR blancaEspecular(1.0f, 1.0f, 1.0f, 1.0f);
```

Nota: Los valores de alfa en la clase D3DXCOLOR son ignorados cuando son usados para describir los colores de la luz.

3.5.2 Materiales

El color de un objeto que vemos en el mundo real es determinado por el color de la luz que este refleja. Por ejemplo, una bola roja es roja porque absorbe todos los colores de luz menos el rojo. La luz roja es reflejada por la bola y llega a nuestros ojos, y por esto vemos la bola como roja. Direct3D modela este mismo fenómeno haciendo que definamos un material para un objeto. Este material nos permite definir el porcentaje al cual la luz es reflejada de la superficie. En código un material esta representado con la estructura D3DMATERIAL8:

```
Typedef struct _ D3DMATERIAL8 {
        D3DCOLORVALUE Diffuse, Ambient, Specular, Emissive;
        Float Power;
} D3DMATERIAL8;
```

La componente Emissive es añadido para aumentar la luz de la superficie, para que parezca que da su propia luz.
Power especifica la nitidez de los resaltes o brillos, cuanto más alto sea el valor, más pequeños, pero también más brillantes.
Como ejemplo, supongamos que queremos una bola roja. Definiríamos el material de la bola para que refleje sólo la luz roja y absorba los demás colores.

```
D3DMATERIAL8 rojo;
ZeroMemory(&rojo, sizeof(rojo));
rojo.Diffuse = D3DXCOLOR(1.0f, 0.0f, 0.0f, 1.0f); //rojo
rojo.Ambient = D3DXCOLOR(1.0f, 0.0f, 0.0f, 1.0f); //rojo
```

```
rojo.Specular = D3DXCOLOR(1.0f, 0.0f, 0.0f, 1.0f); //rojo
rojo.Emissive = D3DXCOLOR(0.0f, 0.0f, 0.0f, 1.0f); //sin color emisivo
rojo.Power = 5.0f;
```

Aquí los valores de verde y azul son 0, indicando que el material no refleja ninguno de estos colores, y el valor de rojo a 1 indicando que refleja 100% luz roja. Nótese como tenemos la habilidad de controlar el color de luz reflejado para cada tipo de luz. También cabe mencionar que si definimos una luz azul, no iluminaría la bola pues absorbería todo ese color. Un objeto aparece negro cuando absorbe toda la luz y aparece blanco cuando la refleja toda.
La estructura de un vértice no tiene la propiedad del material, en cambio, un material debe ser establecido. Para hacerlo hacemos el siguiente llamado:

```
IDirect3DDevice8::SetMaterial(CONST D3DMATERIAL8* pMaterial);
```

Supongamos que queremos dibujar diferentes objetos con distintos materiales cada uno, entonces lo haríamos así:

```
lpd3ddevice8->SetMaterial(&MaterialAzul);
dibujarObjeto(); //objeto azul
lpd3ddevice8->SetMaterial(&MaterialRojo);
dibujarObjeto(); //mismo objeto pero rojo
```

Hay que tener en cuenta de que si no vamos a querer seguir utilizando un material y lo hemos establecido tenemos que volver a poner un material por defecto. Si no lo hacemos podemos encontrarnos con efectos de dibujo que no deseamos.

La normal de una cara es un vector que describe la dirección a la cual un polígono está apuntando. Las normales de los vértices están basadas en la misma idea, pero en vez de hacerse por polígono, las especificamos para cada vértice que forma el polígono, como se muestra en la figura 3.8.

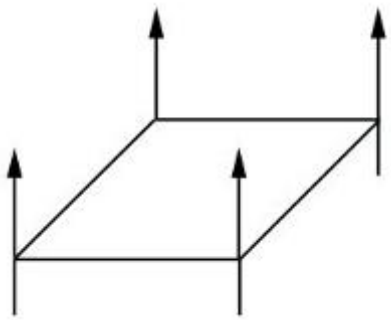


Figura 3.8 El dibujo muestra las normales de cada vértice de un polígono.

Direct3D necesita conocer las normales de los vértices para poder determinar el ángulo al cual la luz le llega, y como la iluminación se hace por vértice, hay que enviar la información para cada vértice. Cabe notar que la normal de un vértice no es necesariamente la normal de la cara a la cual pertenece. Las esferas son buenos ejemplos de objetos cuyas normales de los vértices no coinciden con las de las caras.

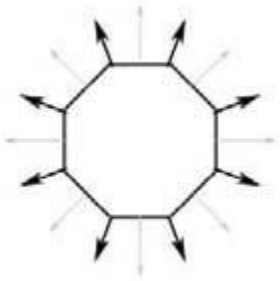


Figura 3.9 Normales de los vértices de un polígono que representa una circunferencia.

3.5.3 Fuentes de luz

Direct3D tiene 3 tipos de luz: Luces puntuales, luces direccionales y luces focales. Para que las luces funcionen hay que habilitarlas primero:

lpd3ddevice8->SetRenderState(D3DRS_LIGHTING, 1);

3.5.3.1 Luces Puntuales (Point Lights)

Esta fuente de luz tiene una posición en el mundo y emite luz en todas las direcciones.

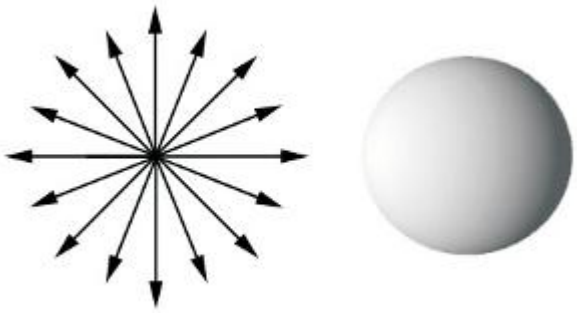


Figura 3.10 Luz puntual.

3.5.3.2 Luces Direccionales (Directional Lights)

Esta fuente de luz no tiene posición, pero emite rayos de luz paralelos a la dirección especificada.

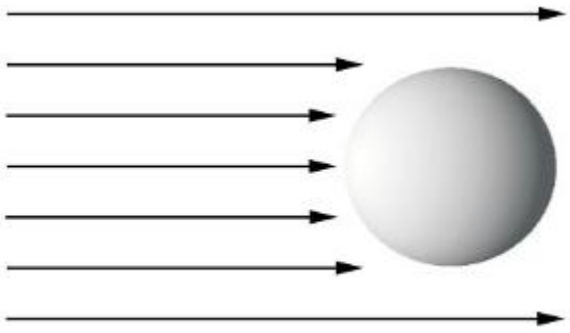


Figura 3.11 Luz direccional.

3.5.3.3 Luces Focales (Spot Lights)

Este tipo de fuente es similar a una linterna. Tiene una posición y brilla de forma cónica en una dirección particular. El cono se caracteriza por dos ángulos, uno interno (theta) que describe cuando empieza a decaer a luz, y uno externo (phi) que describe hasta donde se emite luz.

Figura 3.12 Luz focal.

3.5.3.4 Representación de la luz

En código una fuente de luz esta representada con la estructura D3DLIGHT8:

```
Typedef struct _D3DLIGHT8 {
        D3DLIGHTTYPE Type;
        D3DCOLORVALUE Diffuse;
        D3DCOLORVALUE Specular;
        D3DCOLORVALUE Ambient;
        D3DVECTOR Position;
        D3DVECTOR Direction;
        float Range;
        float Falloff;
        float Attenuation0;
        float Attenuation1;
        float Attenuation2;
        float Theta;
        float Phi;
} D3DLIGHT8;
```

Type Define el tipo de luz que estamos creando: D3DLIGHT_POINT, D3DLIGHT_SPOT, D3DLIGHT_DIRECTIONAL.
Diffuse El color de la luz difusa que esta fuente de luz emite.
Specular El color de la luz especular que esta fuente de luz emite.
Ambient El color de la luz ambiente que esta fuente de luz emite.
Position Un vector describiendo la posición en el mundo de la fuente de luz. Este valor no se usa con luces direccionales.
Direction Un vector que nos dice hacia donde apunta la luz. No se usa para luces puntuales.
Range Indica hasta dónde puede viajar la luz antes de disiparse. No se usa en las luces direccionales.
Falloff Este valor es usado sólo con las luces focales. Define como disminuye la intensidad de luz del cono interior al exterior. Generalmente se deja como 1.0f.

Attenuation0, Attenuation1, Attenuation2 Estas tres variables definen como la luz decrece con la distancia y no se utiliza con luces direccionales. La variable **attenuation0** define una disminución constante, **attenuation1** define una disminución linear, y **attenuation2** define una disminución cuadrática de la luz. Se calcula con la siguiente fórmula, siendo D la distancia hasta la fuente de luz y A0, A1 y A2 corresponden a la atenuación representada por la ecuación 3.3.

$$Atenuación = \frac{1}{A_0 + A_1 \cdot D + A_2 \cdot D^2}$$ Ecuación 3.3

Tetha Usado para las luces focales, especifica el ángulo interno del cono de luz.
Phi Angulo externo del cono de luz de las luces focales.

Una vez hemos creado una luz, necesitamos registrarla con una lista interna de luces que Direct3D mantiene. Lo hacemos de la siguiente manera:

```
lpd3ddevice8->SetLight( 0,   //número que tendrá la luz como identificador
                        &Light); //estructura que tiene la luz definida previamente
```

Una vez registrada, la podemos encender y apagar usando el siguiente método:

```
lpd3ddevice8->LightEnable( 0, //identificador de la luz que queremos
                                //encender/apagar
                           true); //true=encender y false=apagar
```

3.6 PLANO DE AGUA

Para hacer más completo el escenario optamos por añadirle un plano que representaba agua. Este plano de agua sería independiente del terreno que tuviéramos.

3.6.1 Características del plano de agua

Para calcular los parches visibles del plano de agua se ha utilizado un árbol cuaternario que se creaba al inicializar el plano. Este árbol contiene en cada nodo un bloque (estos bloques tienen 4 hijos) y en sus hojas contiene parches del plano de agua. Cada bloque tiene asociado una caja de colisiones para realizar el corte de los parches que se ven según la posición de la cámara. El método es rápido y da buenos resultados debido a que si un bloque se detecta que no se tiene que ver ya no seguimos por ese camino y nos quitamos muchos casos por estudiar.

3.6.2 Implementación

Para implementar el plano de agua optamos por tratarlo como si fuera otro terreno pero cuyos vértices tuvieran una altura constante. Superponiendo el terreno y el plano de agua obteníamos un escenario con lagos en aquellos lugares donde la altura del agua era mayor que la altura del terreno.

Esta solución no era eficiente en el sentido de que podía haber demasiados vértices del plano del agua que no se vieran por estar por debajo del terreno pero que sin embargo se dibujaban con el correspondiente retardo de tiempo. Esto se debía a que el algoritmo de recorte tenía en cuenta sólo la posición de la cámara, no si la altura del terreno era mayor que el agua. La primera mejora que tuvimos que hacer fue la de eliminar los sectores del plano de agua que no se vieran por estar por debajo del terreno.

3.6.3 Eliminación de parches no visibles

Para realizar esta mejora se recorría el árbol cuaternario en su creación y al llegar a las hojas se miraba si la altura del plano de agua era mayor que la altura mínima del terreno en esa zona. Debido a esto el terreno ya no es independiente del agua, deben tener las mismas dimensiones y poder interactuar. En este caso le pasamos al agua como parámetro el terreno y el agua le preguntaba al terreno cada vez que llegaba a una hoja cuál era su altura mínima en dicha posición. Si la altura del agua sale mayor, la hoja se deja como está. En caso contrario borramos dicha hoja. Es decir, hacemos una poda del árbol con las ramas que no se ven nunca por estar por debajo.

Ahora cada nodo del árbol no tiene que tener obligatoriamente 4 hijos, puede haber nodos que tengan menos de 4 hijos.

Estaba solucionado el problema del dibujo del plano de agua, pero teníamos un plano cuya agua estaba siempre quieta. El siguiente paso fue dotar al plano de agua de ecuaciones físicas que simularan olas en la superficie del agua (Ver sección "Física de la implementación").

3.6.4 Archivos de efectos

Para dotar al agua de mejores efectos visuales hemos usado scripts de efectos de Direct3D.

3.6.4.1 Script de efectos

Un script de efectos es un conjunto de instrucciones que le indican a Direct3D como realizar diversas técnicas que puede usar para dibujar un efecto. Los efectos tratan sobre las texturas del dibujo, la iluminación, el color, etc. Le dan al programador mucho control sobre el dibujo de los objetos.

Algunas de estas técnicas tienen muy buenos resultados pero sólo funcionan con tarjetas gráficas potentes. Como no todas las tarjetas soportan estas técnicas sólo se carga el archivo de efectos si se le indica así en el archivo de configuración del terreno que lee el programa a través de LUA.

El proceso a seguir para usar los efectos son:

1. Crear un archivo de efctos. El archivo de texto contendrá técnicas de dibujo. Cada técnica tendrá una o más pasadas para dibujar.
2. Para empezar a usar el archivo hay que llamar a la función D3DXCompileEffectFromFile, esta función cargará el archivo y lo compilará en una interfaz ID3DXBuffer. También hay variantes para obtener los efectos desde una posición de memoria en vez de un archivo.
3. Llamar a la función D3DXCreateEffect que toma una interfaz ID3DXBuffer y crea una interfaz ID3DXEffect que contiene todas las técnicas.
4. Llamar a la función GetTechnique de la interfaz ID3DXEffect para obtener una técnica. Podemos validar la técnica llamando al método Validate.
5. Para usar la técnica cargada, llamamos a la función Begin de la interfaz ID3DXTechnique después de haber llamado a la función BeginScene. Begin devuelve el número de pasadas que debemos efectuar para que la técnica funcione. Tenemos que crear un bucle con dicho número de pasadas, en cada pasada llamaremos al método Pass de la interfaz ID3DXEffect y luego llamaremos a los métodos de dibujo como lo hacemos normalmente.

Las dos cosas más importantes a recordar es llamar a la función Begin para poder llamar al método Pass cada vez que entremos en una nueva pasada de la técnica a usar.

3.6.5 Vista debajo del agua

Una vez que tuvimos el plano de agua funcionando, queríamos que cuando se estuviera debajo del agua se notara el cambio. Según lo realizado hasta el momento estando debajo del agua se veía el terreno igual que si no hubiera agua.

3.6.5.1 Texturas

La solución que creamos para este problema fue el de poner una textura semitransparente de agua en la cámara, así todo lo que se viera a través de la cámara aparecería como si lo viéramos a través de una especie de cristal azul. Para dibujar la textura primero comprobamos la altura de la cámara con la altura del agua en ese punto, y si la cámara está por debajo del agua se dibujaba la textura en toda la pantalla.

3.6.5.2 Añadiendo profundidad

Con la solución realizada hasta el momento se obtuvieron buenos resultados pero todavía quedaba algo colgando. Cuando estás debajo del agua la luz tiene que ser menos intensa, es decir, las cosas se tienen que ver más oscuras. Y si queremos ser más precisos a medida que te vayas hundiendo más se debería ver más y más oscuro.

La primera solución que se pensó fue en utilizar luces y oscurecerlas cuando se estuviera debajo del agua, pero los resultados no fueron satisfactorios. Otra opción, que fue la que hemos elegido, era oscurecer la textura a medida que nos hundimos más. La solución es más sencilla y rápida que la anterior y los resultados visuales son mejores.

Para realizar todo esto hay que jugar un poco con los estados de renderización de DirectX. Tenemos que establecer unos valores concretos:

SetLighting(false): indicamos que no se tenga en cuenta la luz.

SetAlphaBlendEnable(true): Indicamos que la textura va a poder mezclarse, va a tener transparencia.

SetSrcBlend(D3DBLEND_SRCALPHA): Le indicamos que la transparencia depende del valor de alpha que tenga el color de los vértices. Si el valor de alpha es próximo al cero, la textura será más transparente.

SetDstBlend(D3DBLEND_SRCCOLOR): Le indicamos que la textura va a tener un destino igual al color que tengan los vértices. Si el color de los vértices está cerca del cero, los colores de la textura serán más oscuros.

En el programa dejamos siempre un valor de alpha de 255 (ff) y el color lo vamos variando según la diferencia de alturas de la cámara y el agua. Si la diferencia es mayor de 100 el color es igual al negro: 0xff000000. Cuanto menor sea la diferencia más nos acercaremos al valor del blanco 0xffffffff. El color se codifica como: 0xaarrggbb, siendo aa el valor de alpha, rr la cantidad de rojo, gg la cantidad de verde y bb la cantidad de azul. El valor de rr,gg y bb lo modificamos igual en los tres, es decir, si rr=a8 entonces gg = bb = a8. Así conseguimos oscurecer o aclarar todos los colores en la misma cantidad.

3.7 CIELO

Una forma de crear imágenes de paisaje de fondo para programas en 3D es crear un cubo bastante grande, y poner la cámara justo en el centro del cubo.

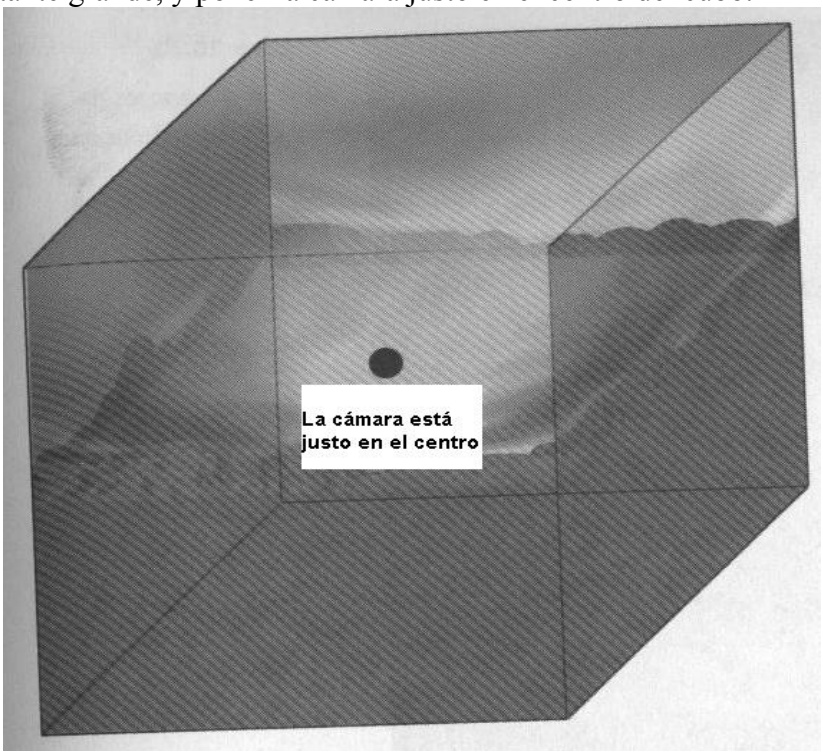


Figura 3.13 Dibujo de una "skybox".

3.7.1 Skybox

Este cubo es el llamado skybox. Por dentro de las caras del cubo pintamos las imágenes de fondo que queremos que se vean, necesitamos 6 imágenes si contamos la cara del cubo que hace de suelo. El truco de este método consiste en mover el cubo con la cámara. Así, aunque la cámara se mueva la imagen del cubo (el horizonte) permanece siempre a la misma distancia.

Como mejora del cielo optamos por implementar tres tipos de cielo: SkyBox, Dome y SkyStrip que usan técnicas diferentes de dibujo. Cada técnica se usa estableciendo los vértices a dibujar de una determinada forma, dependiendo del tamaño del cubo.

SkyBox. Usa cinco texturas, no cargamos la textura del suelo. Para dibujarlas utiliza el método D3DPT_TRIANGLEFAN.

SkyStrip. Utiliza también cinco texturas que dibuja utilizando D3DPT_TRIANGLESTRIP.

Dome. Usa dos texturas una para el cielo y otra para las nubes que se superponen. Esta técnica en vez de dibujar una caja, dibuja una esfera. La idea es la misma que con la caja, pero cambian los vértices. Esta técnica usa muchos más vértices que las dos anteriores.

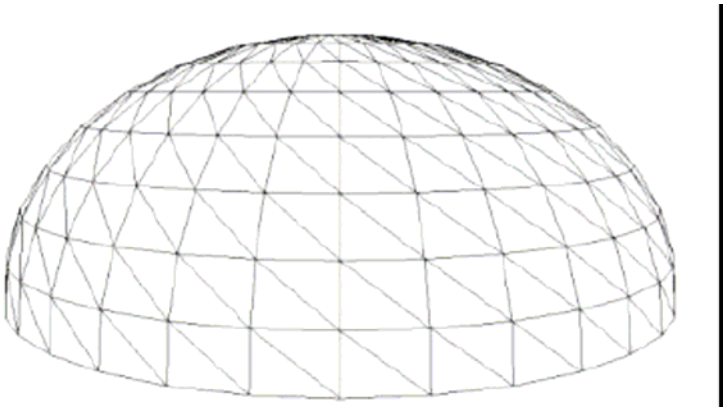


Figura 3.14 Dome

TriangleStrip: Indica una tira de triángulos, cada triángulo comparte dos de sus vértices con su vecino. De esta forma se ahorran vértices.

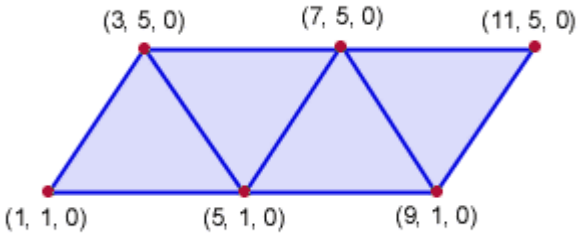


Figura 3.15 TriangleStrip.

TriangleFan: Indica un conjunto de triángulos que comparten un vértice entre todos.

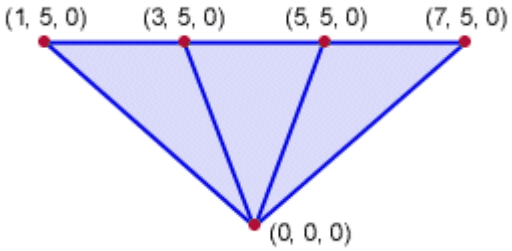


Figura 3.16 TriangleFan

Para que el cielo se vea correctamente hay que tener en cuenta que el cubo que queremos crear no sea mayor que el plano de corte lejano de la cámara, porque sino no se verán las paredes del cubo al estar demasiado lejos. Hay que establecer como máximo un cubo de dimensiones igual al plano lejano.

3.8 DESTELLOS DEL SOL

Los efectos del sol los hemos implementado a base de texturas. Dependieno de la posición del sol y de la cámara dibujamos las texturas que simulan destellos en una posición u otra. La idea es que tenemos un vector desde la posición de la cámara hasta el sol. Si los destellos se tienen que ver, es decir, si vemos el sol, calculamos las posiciones adecuadas para cada destello en ese vector, y las dibujamos como cuadrados que miran a la cámara (Ver Billboarding). Estas texturas tienen transparencia.

3.8.1 Visibilidad de los destellos

Para determinar si el sol se ve o no, usamos un rayo que va desde la cámara hasta el sol. Hay que comprobar dos tipos de intersección. La primera si dicho rayo incide sobre algún objeto de la escena el sol no se verá. Para comprobar si el rayo incide sobre un objeto en cuestión llamamos al método IntersectRay(origen, dirección) de la caja que contiene dicho objeto. El método de la clase AABBox iterará empezando con la caja más general del objeto mientras haya subcajas. La idea es que el objeto se divide en partes más pequeñas que tienen cada una caja más pequeña que la anterior. Si llegamos a una caja de una hoja del árbol de cajas y nos sale que hay intersección, entonces el rayo incide sobre el objeto y el sol no debe verse. La segunda es si incide sobre el terreno, es decir, que el sol esté tapado por alguna montaña, por ejemplo. Para esto usamos también la misma técnica que antes. El terreno está dividido en cajas. Se compone de un árbol cuaternario. Primero comprobamos si hay intersección con alguna caja que pertenezca a alguna hoja del árbol cuaternario, Si es así, comprobamos si el rayo incide sobre algún polígono que compone el parche perteneciente a esa hoja y si incide, el sol no debe verse.

3.8.2 Intensidad del sol

Para hacer más realista el sol, cuando lo miramos directamente la intensidad aumenta, si lo miramos con más desviación la intensidad disminuye. Así, la luz no desaparece de golpe, sino que tarda un poco en desvanecerse.

4. TERRENO

Esta sección trata la creación de terrenos en 3D. Tratamos los mapas de alturas por imágenes o valores numéricos y terrenos aleatorios. Se aborda también la generación de terrenos infinitos y el suavizado de las alturas de un terreno.

4.1 MAPAS DE ALTURAS

El terreno se puede leer desde un archivo que tenga un mapa de alturas o bien generar el terreno aleatoriamente. Las alturas pueden estar representadas mediante floats, o bien mediante una imagen de tipo RAW. Todos los terrenos tiene una escala asociada para indicar la distancia entre puntos de la malla que va a representar el terreno, así como una opción de suavizado de bordes que se activa según el archivo de configuración “Terreno.Lua”.

4.1.1 Archivo de alturas

Un mapa de alturas puede estar representado por un archivo que contiene números de alturas.

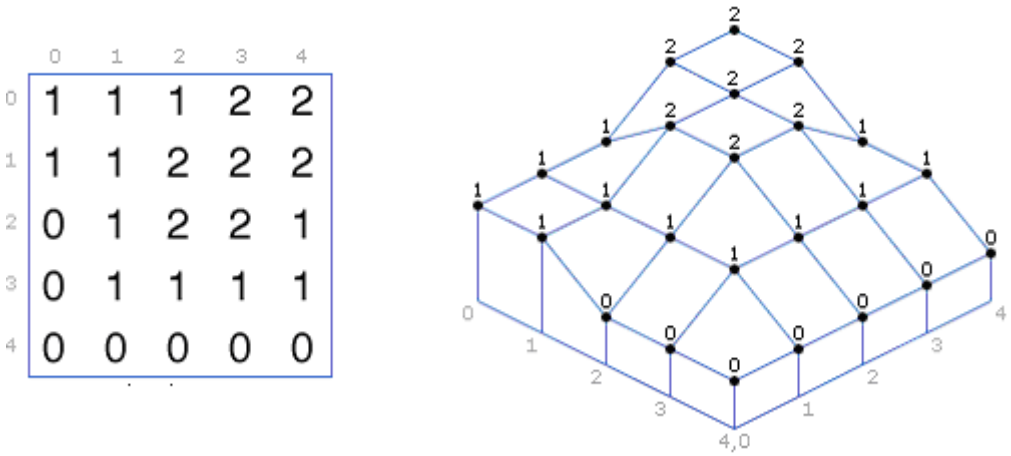


Figura 4.1 A la izquierda contenido de un archivo de alturas, a la derecha representación del terreno según dichas alturas.

Para leer este tipo de archivo lo abrimos y almacenamos su contenido en un array de floats. Recorriendo dicho array obtenemos las alturas del terreno en cada punto.

4.1.2 Alturas según una imagen

Otra forma de ver un mapa de alturas es pensar en una imagen en escala de grises, donde los puntos más claros son los de mayor altura:

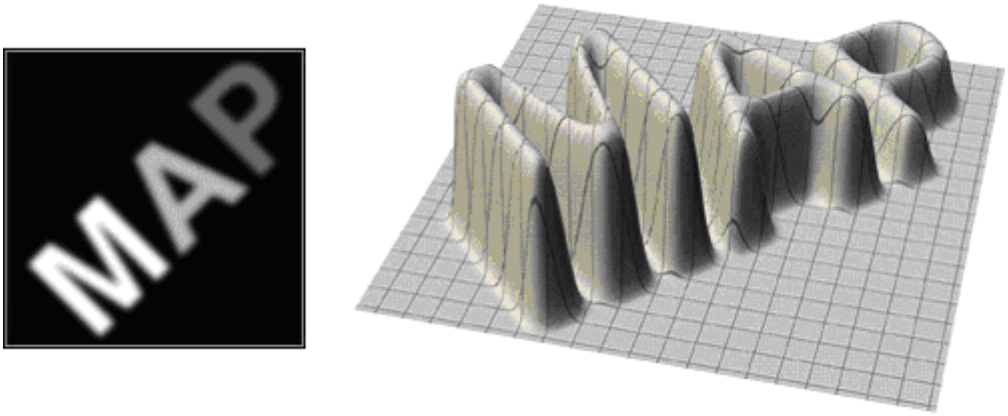


Figura 4.2 A la izquierda imagen que representa las alturas, a la derecha el terreno representado a partir de dicha imagen.

Para leer este tipo de archivos primero leemos la imagen y cargamos los datos. Hay que comprobar si el archivo está comprimido y que formato tiene. Una vez que tenemos los datos cargados se opera como en el caso anterior, esta vez los datos son de tipo char.

4.2 TERRENO ALEATORIO

La otra alternativa del terreno es generar alturas aleatorias. Una forma de crear terreno aleatorio es el algoritmo Hill.

4.2.1 Algoritmo Hill

Este algoritmo se basa en lo siguiente:

1. Empieza con un terreno liso.
2. Coge un punto de forma aleatoria dentro del terreno o cerca de él, y un radio de forma también aleatoria entre un mínimo y un máximo prefijados. Dependiendo de este mínimo y máximo el terreno será más o menos escarpado.
3. Levanta un montículo en el punto determinado con el radio escogido.
4. Volver al paso 2 e iterar las veces que se hayan estipulado. El número de iteraciones también interviene en el aspecto del terreno.
5. Normalizar el terreno.
6. Alisar las bases de los montículos.

4.2.1.1 Montículos (Paso 3)

Nos vamos a centrar ahora en el punto 3. Un montículo es una especie de semiesfera. Cuanto más grande sea el radio, mayor altura tendrá el montículo. Matemáticamente hablando es una parábola. Dado un centro (x1,y1) y un radio r, la altura z en el punto (x2,y2) es equivalente a la ecuación 4.1.

$$z = r^2 - ((x_2-x_1)^2 + (y_2-y_1)^2)$$

Ecuación 4.1

La ecuación 4.1 genera montículos de la forma mostrada en la figura 4.3.

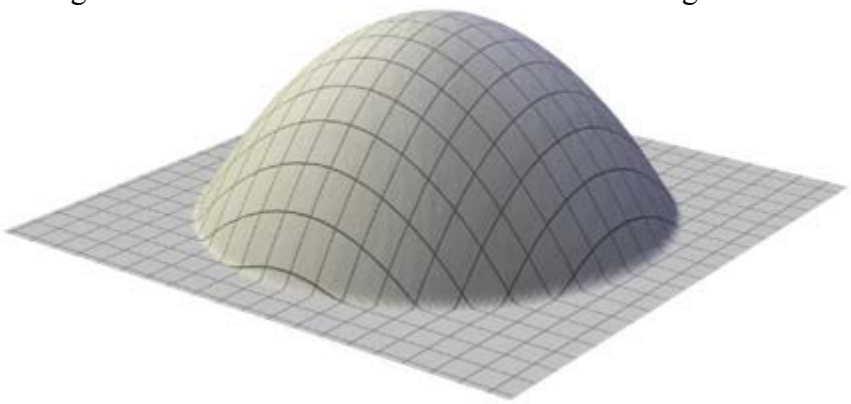


Figura 4.3 Montículo generado por la ecuación 4.1

Para generar un terreno completamente hay que iterar la construcción de montículos. Cuando dos montículos se superponen, los vértices superpuestos tendrán como altura la suma de los dos.

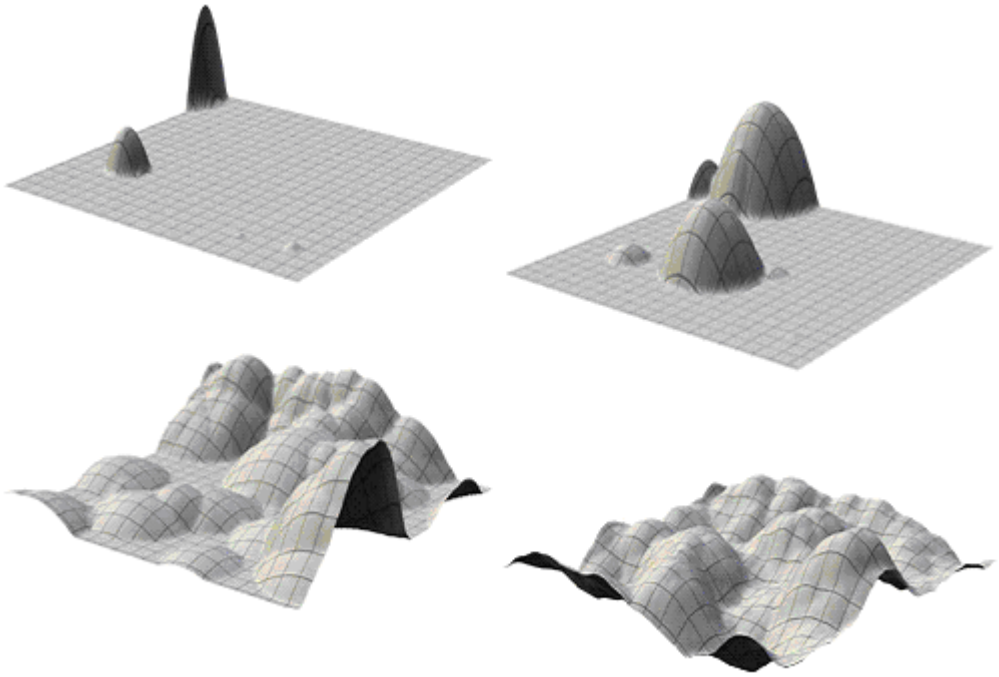


Figura 4.4 El dibujo de la parte superior izquierda muestra un terreno creado con 5 iteraciones, el de la parte superior derecha con 10 iteraciones, el terreno de la parte inferior izquierda tiene 50 iteraciones y el de la parte inferior derecha está hecho con 200 iteraciones.

4.2.1.2 Normalización del terreno (Paso 5)

Normalizar el terreno significa establecer los valores de las alturas entre 0 y 1. De esta forma el terreno se puede escalar como queramos. Como se genera de forma aleatoria no podemos asegurar que los valores mínimo y máximo estén entre 0 y 1. Para normalizarlo necesitamos conocer el máximo y el mínimo de todas las alturas, una vez que los tengamos operamos según la ecuación 4.2.

$$z_{norm} = \frac{z - min}{max - min}$$

Ecuación 4.2

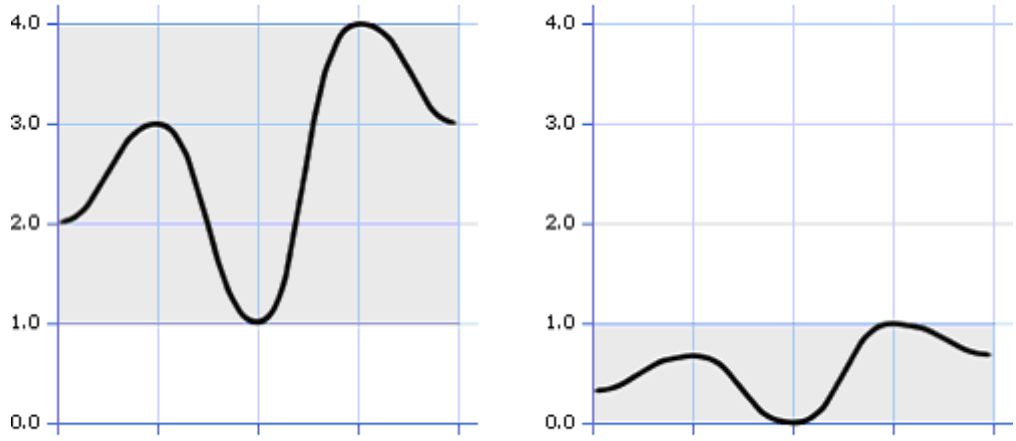


Figura 4.5 A la izquierda se muestran las alturas del terreno sin normalizar, a la derecha las alturas después de normalizar.

4.2.1.3 Suavizado de las bases (Paso 6)

El terreno generado hasta ahora necesita modificar un poco la forma en que los montículos llegan al suelo para no crear pendientes muy pronunciadas que terminen de golpe. Para ello alisamos la base de los montículos.

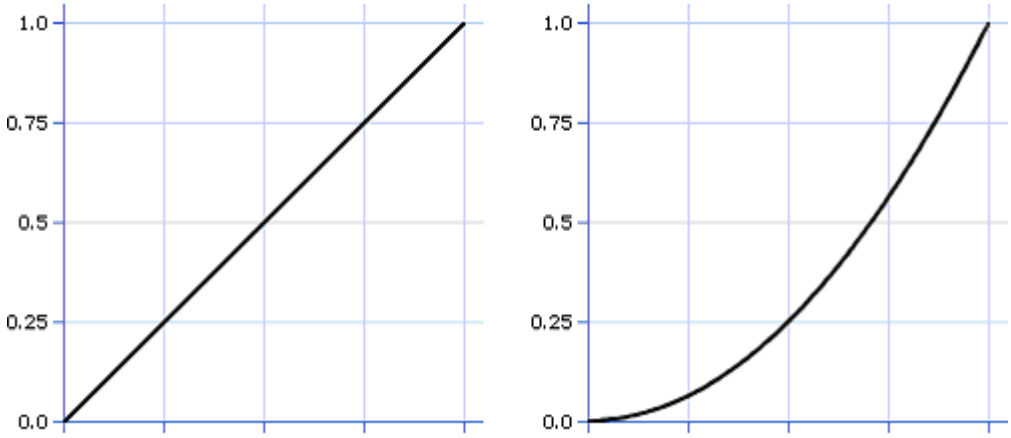


Figura 4.6 El dibujo de la izquierda representa las alturas sin alisar, el de la derecha las alturas suavizadas.

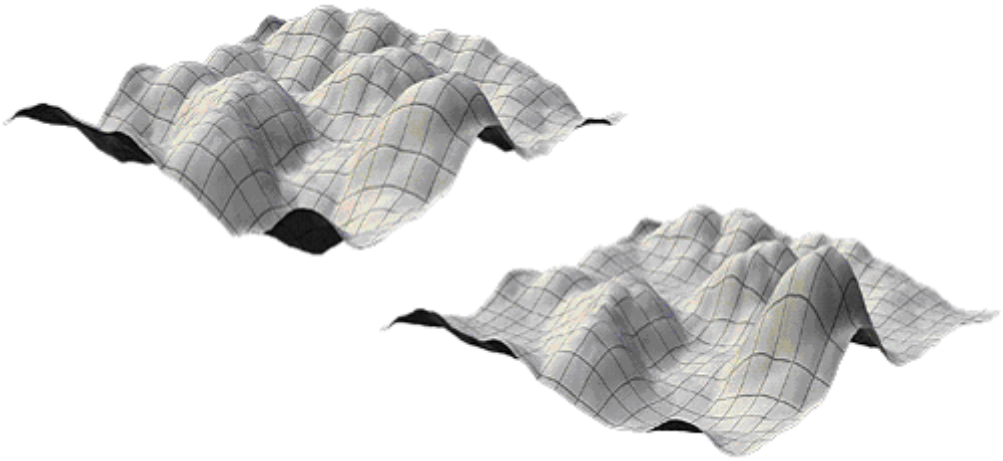


Figura 4.7 El dibujo de la parte superior izquierda representa el terreno sin alisar, el de la parte inferior derecha muestra el terreno alisado.

4.2.2 Mejora de los terrenos aleatorios

Para poder generar terrenos cuyos montículos no se salgan de los límites del terreno tenemos que tener en cuenta unos detalles. Necesitamos dos valores aleatorios, distancia y theta. La distancia será cuánto nos podemos alejar del centro del terreno. Podrá valer desde 0 hasta la mitad de la longitud del mapa menos el radio del montículo. Esto previene de alcanzar los límites del mapa. Theta determina en qué dirección a partir del centro se situará el montículo. Podrá valer desde 0 hasta dos pi. Con estos dos valores podemos calcular la x y la y del centro del montículo y proceder con el algoritmo anterior. Para generar los puntos centrales de los montículos usamos las ecuaciones 4.3.

$$x = \frac{longitud}{2} + \cos(\theta) \cdot dis\tan cia$$
$$y = \frac{longitud}{2} + \sin(\theta) \cdot dis\tan cia$$

Ecuación 4.3

La figura 4.8 muestra los resultados de esta mejora.

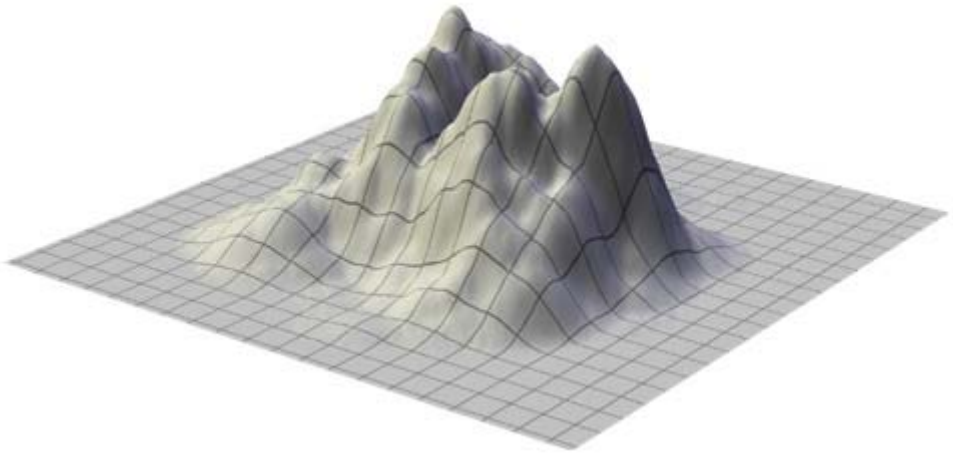


Figura 4.8 Terreno cuyos montículos no se salen de sus límites.

Con este tipo de terrenos se puede conseguir un buen efecto si añadimos agua.

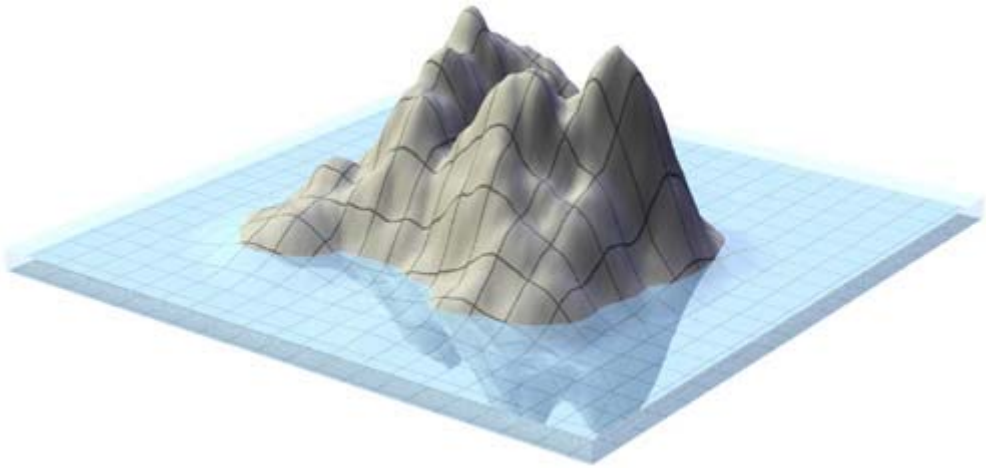


Figura 4.9 Terreno con un plano de agua.

4.3 SUAVIZADO DE BORDES

Para suavizar los bordes hemos usado una técnica de filtrado de cajas. Éste es un método que reduce la intensidad de variación entre pixels de una imagen, y se usa normalmente para eliminar ruidos.

Este tipo de filtrado se basa en la sustitución de cada píxel por una media de sus vecinos, incluido él mismo. El filtrado se basa en un kernel. Este kernel representa el área que rodea al píxel. El kernel más usual es de 3x3. Se pueden usar kernels de dimensiones mayores y hacer menos iteraciones o bien usar un kernel más pequeño y hacer más iteraciones. El kernel que hemos utilizado tiene dimensiones 3x3 y está representado en la figura 4.10.

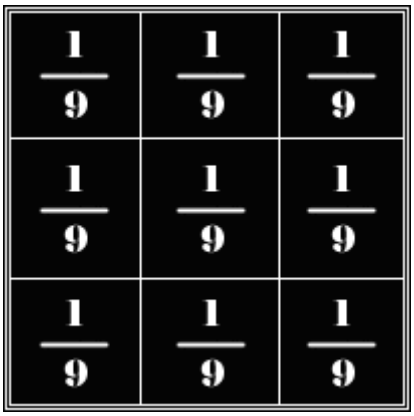


Figura 4.10 Kernel de filtrado de 3x3.

Con este tipo de filtrado eliminamos bordes muy pronunciados.

4.4 NIVEL DE DETALLE

Tanto en el terreno como en el plano de agua se ha hecho un sistema que tiene en cuenta la distancia a la que está un determinado parche para dibujarlo de una forma más simple o menos simple. En cualquier caso, si estamos muy cerca del parche, éste se dibujará con el nivel de detalle máximo.

4.4.1 Árbol cuaternario

Vamos a explicar primero el "quad tree". En nuestro programa usamos un array para guardar las alturas del mapa en cada punto. Para dibujar el terreno usamos un "quad tree". Un "quad tree" divide el terreno recursivamente en bloques de 4. Cada bloque almacena el punto más alto y el punto más bajo que contiene. Cuando dibujamos, si podemos determinar que un bloque está completamente escondido podemos descartar inmediatamente a todos sus hijos. El "quad tree" nos permite descartar rápidamente polígonos escondidos y evitar dibujarlos.

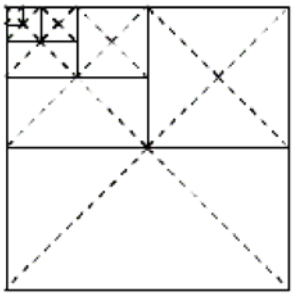


Figura 4.11 Posible recorrido de un árbol cuaternario

4.4.2 Métrica del nivel de detalle

Ahora explicaremos la métrica usada para el nivel de detalle. Para cada parche de la malla del terreno guardamos el error máximo permitido y con él calculamos la distancia a la cual se debe dibujar el parche con un nivel de detalle determinado, habrá tantos niveles de detalle como hayamos indicado en la implementación. Cuando vamos a dibujar un parche calculamos una métrica basada en lo siguiente. Primero calculamos la posición relativa de la cámara y el punto central del parche. Nuestra métrica será la coordenada máxima de dicho punto:

```
Point3 posRel=vPos-cameraPos;
float L1=max(max(fabs(posRel.x()),fabs(posRel.y())),fabs(posRel.z()));
```

Donde vPos es el punto central del parche. Para cada nivel de detalle del parche comprobamos si L1 es mayor que la distancia de dicho nivel, si L1 es mayor no dibujamos más niveles, si es menor, aumentamos en 1 el número de niveles a dibujar. Con este sistema nos podemos ahorrar dibujar vértices que se pueden quitar teniendo un nivel de detalle menor según la distancia, es muy útil a la hora de dibujar montañas lejanas. Al usar la posición relativa nos evitamos tener que calcular la distancia real, y así hacer cálculos más complicados como la raíz cuadrada.

4.5 TERRENO INFINITO

En el simulador inicialmente teníamos un terreno de dimensiones determinadas, así que podíamos salirnos del terreno y continuar en cualquier dirección. El efecto de esto no era muy bueno porque permitía ver el terreno desde abajo y el efecto no era de nuestro agrado. Para solucionarlo pensamos en crear un terreno autogenerado, de manera que siempre estuviéramos sobre un terreno. Pensamos en el terreno como si fuera un donut, si nos salíamos por un lado entraríamos en otro terreno igual que el anterior pero por el lado contrario.

4.5.1 1ª Aproximación: Modificar el terreno dinámicamente

La primera aproximación fue intentar ir cambiando el terreno sobre la marcha, de manera que el objeto que manejamos siempre estuviera en el centro del terreno visible, y a medida que avanzara el objeto el terreno se fuera actualizando. De esta forma al avanzar en una sección del terreno, la última sección del mismo pasaba a ser la primera.

Este camino de hacer las cosas pronto empezó a dar problemas. El terreno está organizado según un árbol cuaternario, dicho árbol almacena en los nodos el punto máximo y el punto mínimo del terreno en la sección que guarda dicha parte del árbol. Al ir cambiando las secciones del terreno teníamos que ir cambiando también el árbol, construyéndolo otra vez desde el principio, ya que al cambiar una sección modificamos toda la estructura del árbol.

4.5.2 2ª Aproximación: Actualizar la posición del objeto

El siguiente camino a tomar fue el de actualizar la posición del objeto, si llegaba a salirse del terreno lo colocábamos según su posición módulo la longitud del terreno. De esta forma parecía que avanzábamos por otro terreno distinto pero sin embargo era el mismo. Esta aproximación era la más rápida pero no fue de nuestro agrado, porque el objetivo que teníamos era que pudieras ir a cualquier punto, de esta forma sólo nos podíamos mover en las dimensiones del terreno.

4.5.3 3ª Aproximación: Rejilla de terrenos

El siguiente intento fue el de crear una especie de rejilla de terrenos, concretamente nueve, cada terreno con su árbol cuaternario. De esta forma el objeto estaría en el terreno central, rodeado por los ocho restantes. Cuando el objeto pasaba de

un terreno a otro las posiciones de los terrenos se actualizaban para colocar al nuevo terreno en el centro de la rejilla de terrenos. Esta solución da buenos resultados en cuanto a la escena del simulador, sin embargo tener cargados nueve terrenos y poder llegar al punto de dibujarlos a todos supone una carga en el rendimiento inmensa. Se podría haber usado este método si los terrenos fueran pequeños y con pocos triángulos a dibujar, o si qusieramos que el simulador sólo pudiera ejecutarse en ordenadores muy potentes. Pero no era nuestro caso.

4.5.4 4ª Aproximación: Terreno actual y terreno siguiente

Por último pensamos en una solución más simple que la anterior pero que siguiera la misma idea. La escena tendría dos terrenos, uno que sería el terreno donde estuviera el objeto a manejar y otro que sería el siguiente terreno, a donde iría el objeto. Para hacer esto, el terreno auxiliar se construye según la posición del objeto, si el objeto está más próximo a la parte derecha del terreno actual, el nuevo terreno se establece en la parte derecha. De esta forma, vemos el terreno hacia dónde nos dirigimos y cuando nos salimos del terreno actual, se establece como terreno actual el auxiliar y el que era el actual como auxiliar.

Para trabajar con un terreno variable tuvimos que hacer que el objeto a manejar tuviera un terreno como atributo, este terreno sería el terreno activo en cada momento. Cuando cambiamos de terreno actualizamos el objeto con el nuevo terreno. De esta forma las colisiones con el terreno se hacen correctamente. De no ser así, el objeto comprobaría las colisiones con el terreno original y no comprobaría bien las colisiones.

Hay que tener en cuenta que los terrenos se generan a partir de una posición inicial, una coordenada x de origen y una coordenada z de origen. Para desplazar un terreno de un sitio a otro simplemente hay que cambiar dicho origen.

Este mismo sistema se ha usado con los planos de agua, para tener la misma escena nos movamos a donde nos movamos.

Otro problema que nos surgió con el terreno infinito fue el cálculo de los destellos del sol, inicialmente se hacía con el terreno original, de esta forma aunque el sol estuviera tapado por un terreno auxiliar se seguía viendo. Tuvimos que establecer un método de actualizar el terreno activo en cada objeto de la escena. De esa forma cada objeto sabría cuál era el terreno activo en cada momento.

Si no queremos que el terreno sea igual siempre sólo tenemos que cambiar el terreno sobre la marcha. Por ejemplo con el algoritmo de generación de terrenos aleatorios. Si al generar el terreno le ponemos la misma semilla el terreno será igual que antes, así nos aseguramos de que al volver sobre nuestros pasos el terreno está igual que antes.

5. FÍSICA DE LA IMPLEMENTACIÓN

Vamos a tratar sobre las consideraciones físicas que hemos tenido en cuenta para diversos componentes de la escena del simulador.

5.1 FÍSICA DE LOS PROYECTILES

Para hacer la clase más independiente del programa hemos usado el sistema de scripts de LUA. Con LUA leemos distintas propiedades de los proyectiles:

Velocidad inicial de los proyectiles.
Radio de colisión de los proyectiles.
Modelo 3D de los proyectiles.
Textura de los proyectiles.

En la función Update() usamos también LUA para que actualice los proyectiles según las ecuaciones físicas indicadas en el archivo de texto. Así si queremos que nuestros proyectiles no sean afectados por la gravedad, por ejemplo, simplemente cambiando el archivo de texto modificaríamos el juego, todo ello sin necesidad de recompilar el programa.

En nuestro caso hemos optado por usar como ejemplo unas ecuaciones que tienen en cuenta la gravedad y el rozamiento con el aire. La función de posición para un objeto moviéndose a través de un medio que opone resistencia se muestra en la ecuación 5.1.

$$x(t) = x_0 + \frac{g}{k} \cdot t + \frac{k \cdot v_0 - g}{k^2} \cdot (1 - e^{-k \cdot t}) \qquad \text{Ecuación 5.1}$$

Donde k representa la intensidad de la fuerza de oposición del medio.
La velocidad Terminal viene dada por la ecuación 5.2.

$$v_t = \frac{g}{k} \qquad \text{Ecuación 5.2.}$$

Aparte de las opciones de LUA realizamos algunas comprobaciones dentro del programa. Primero comprobamos si el tiempo de vida ha expirado, sino es así comprobamos que no hay colisión con el terreno. Si hay colisión con el terreno guardamos la distancia al punto de colisión. Segundo comprobamos las colisiones con objetos estáticos, utilizando una tabla del terreno para saber en qué sección del terreno tenemos que comprobar (Ver la sección de colisiones). Si hay colisión miramos si la distancia del objeto es menor que la del terreno en el caso de que hubiera, si es así marcamos como colisión con objeto, si no seguimos. Por último comprobamos con los objetos dinámicos y realizamos la misma operación.

De esta forma a parte de comprobar si hay colisión o no, sabemos con qué objeto ha colisionado antes, en el caso de haber colisión. Y sabremos cómo actuar, explotar el objeto, etc…

Siempre y cuando haya colisión o el tiempo de vida expire, el proyectil se borra del array.

5.2 FÍSICA DEL AGUA

Para solucionar el hecho de que la superficie del plano de agua no se moviera se optó por ir modificando las alturas del plano de agua con criterio.

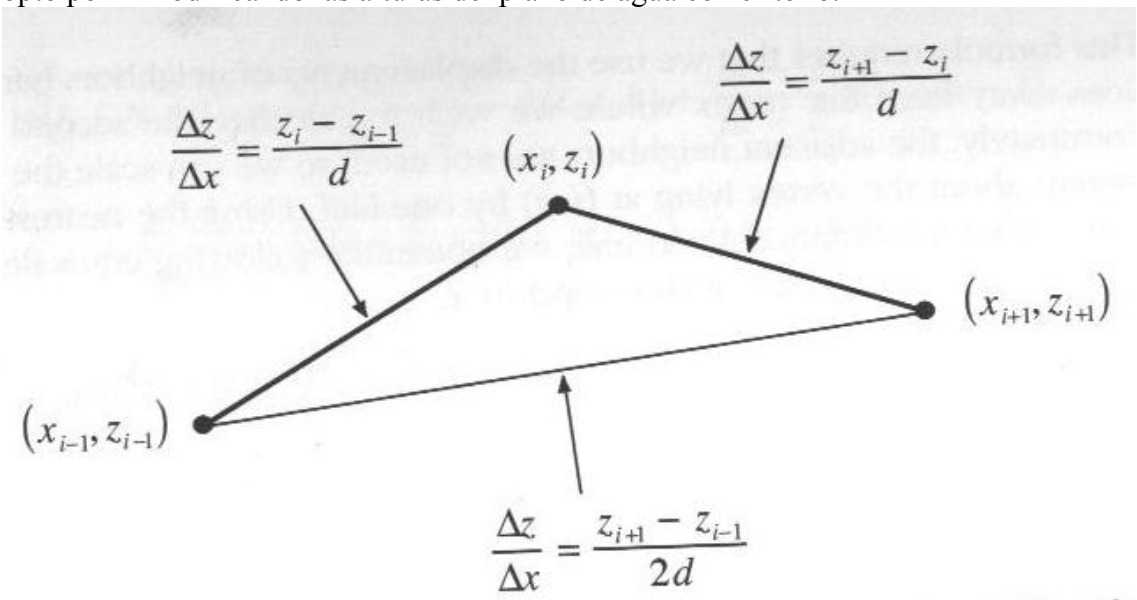


Figura 5.1 Representación de los incrementos en las coordenadas de z con respecto a x.

La ecuación de una ola de dos dimensiones en una superficie que ofrece una resistencia debido a su viscosidad se muestra en la ecuación 5.3.

$$\frac{\partial^2 z}{\partial t^2} = c^2 \cdot \left(\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2}\right) - \mu \cdot \frac{\partial z}{\partial t} \qquad \text{Ecuación 5.3}$$

En la ecuación 5.3 la constante c es la velocidad a la que se propaga la ola a través del medio y la constante μ representa la viscosidad del medio.

La primera derivada de una función $z(x)$ puede aproximarse por la ecuación 5.4.

$$\frac{d}{dx} z(x) = \frac{z(x+d) - z(x-d)}{2 \cdot d} \qquad \text{Ecuación 5.4}$$

Donde d representa una constante de periodo de muestreo.
La segunda derivada puede aproximarse por la ecuación 5.5.

$$\frac{d^2}{dx^2} z(x) = \frac{z(x+d) - 2 \cdot z(x) + z(x-d)}{d^2} \qquad \text{Ecuación 5.5}$$

El desplazamiento futuro $z(i, j, k+1)$ de un punto de la superficie de un fluido después de un tiempo t se calcula usando la ecuación 5.6.

$$z(i,j,k+1) = \frac{4 - 8 \cdot c^2 \cdot t^2 / d^2}{\mu \cdot t + 2} \cdot z(i,j,k) + \frac{\mu \cdot t - 2}{\mu \cdot t + 2} \cdot z(i,j,k-1) + \frac{2 \cdot c^2 \cdot t^2 / d^2}{\mu \cdot t + 2} \left[z(i+1,j,k) + z(i-1,j,k) + z(i,j+1,k) + z(i,j-1,k)\right] \qquad \text{Ecuación 5.6}$$

Donde d es la distancia entre puntos vecinos

.

Dada una constante de tiempo t, la velocidad de la ola c debe satisfacer la ecuación 5.7.

$$0 < c < \frac{d}{2 \cdot t} \sqrt{\mu \cdot t + 2} \qquad \text{Ecuación 5.7}$$

Dada una velocidad de la ola c, el tiempo de muestreo t debe satisfacer la ecuación 5.8.

$$0 < t < \frac{\mu + \sqrt{\mu^2 + 32 \cdot c^2 / d^2}}{8 \cdot c^2 / d^2} \qquad \text{Ecuación 5.8}$$

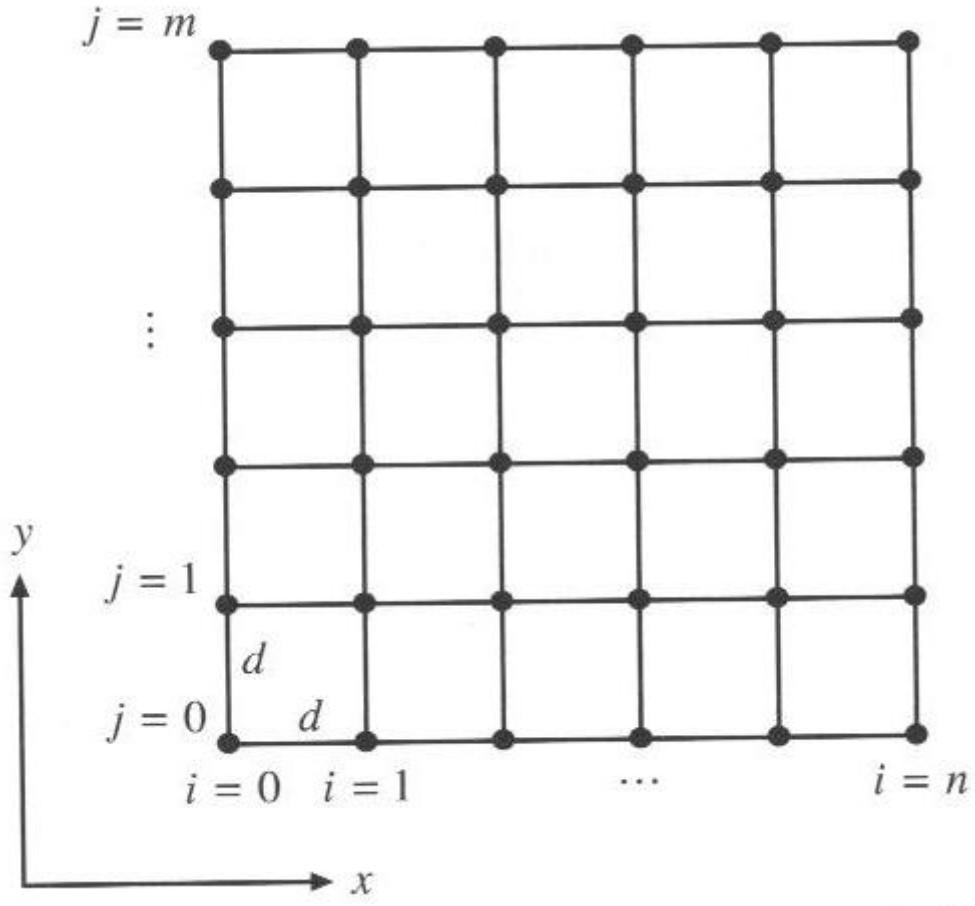


Figura 5.2 En el dibujo se muestra la forma de recorrer los vértices de una malla.

Para actualizar las alturas de cada vértice se implementó un método que recorría todos los parches activos del plano de agua y los actualizaba.

Como casi siempre, esto no era eficiente porque llevaba mucho tiempo actualizar todos los vértices. Se optó por implementar otro método que actualizaba sólo los parches visibles en ese momento. Los parches visibles serían los que se ven dependiendo de la cámara. Con ello nos ahorramos mucho tiempo en la actualización.

Con todo esto teníamos un plano de agua con olas en su superficie, pero las ecuaciones físicas que teníamos tenían en cuenta un período de muestreo constante. Aquí surge un problema, ¿qué pasa si ejecuto el programa en un ordenador más rápido? Cuanto más rápido sea el ordenador, las olas serán más rápidas.

Para solucionar esto se ha optado por utilizar un temporizador de la ventana que siempre le envía un mensaje a la ventana 10 veces por segundo. Teniendo esto, con el método *WndProc()* (ver "Ventana principal") captamos el mensaje del temporizador y actualizamos el agua 10 veces por segundo. De esta forma el agua se mueve a la misma velocidad independientemente del ordenador donde se ejecute el programa.

El fragmento de código dentro del método *WndProc()* encargado del temporizador sería:

```
switch(iMsg) {
        case WM_CREATE://creación de la ventana
                //Establecemos un temporizador de 10 milisegundos
                SetTimer( hwnd, 0, 1000 / 10, NULL );
        break;
        case WM_DESTROY://destrucción de la ventana
                KillTimer( hwnd, 0 );   //eliminamos el temporizador
                PostQuitMessage(0);
                return 0;
        break;
        case WM_TIMER://mensaje del temporizador
                //actualizamos el plano de agua
                water->evalVisible();
        break;
}
```

5.3 FÍSICA RELACIONADA CON EL TERRENO

5.3.1 Fuerzas

A la hora de calcular las fuerzas ejercidas sobre los objetos hemos tenido en cuenta varias fuerzas relacionadas con el terreno. La primera de todas es la fuerza normal. La fuerza normal, reacción del plano o fuerza que ejerce el plano sobre el bloque depende del peso del bloque, la inclinación del plano y de otras fuerzas que se ejerzan sobre el bloque. Supongamos que un bloque de masa *m* está en reposo sobre una superficie horizontal, las únicas fuerzas que actúan sobre él son el peso *mg* y la fuerza y la fuerza normal *N*. De las condiciones de equilibrio se obtiene que la fuerza normal *N* es igual al peso *mg*.

$$N = m \cdot g \qquad \text{Ecuación 5.9}$$

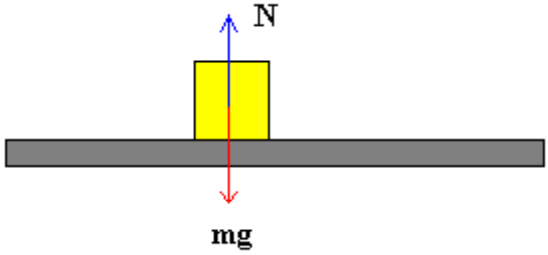


Figura 5.3 Fuerza normal

Si ahora, el plano está inclinado un ángulo θ, el bloque está en equilibrio en sentido perpendicular al plano inclinado por lo que la fuerza normal N es igual a la componente del peso perpendicular al plano.

$$N = mg \cdot \cos\theta$$ Ecuación 5.10

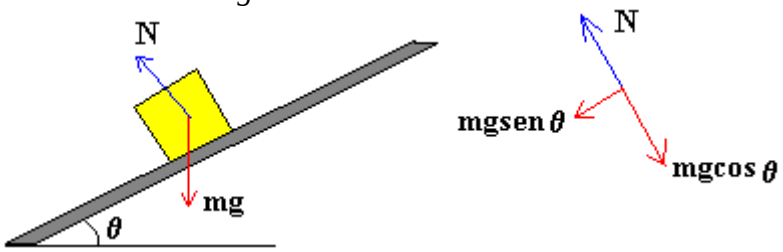


Figura 5.4 El dibujo de la izquierda muestra la fuerza normal y el peso, a la derecha se muestra la descomposición de las fuerzas que actúan sobre el objeto.

Vemos que sobre el bloque de la figura 5.4 aparece una fuerza que es igual al peso por el seno de la inclinación del plano. A la hora de aplicar las fuerzas al objeto hemos tenido en cuenta que nuestros planos son en tres dimensiones, entonces hemos calculado dos fuerzas por separado. Una fuerza vista sobre el plano xy y otra fuerza vista sobre el plano zy.

Consideremos de nuevo el bloque sobre la superficie horizontal. Si además atamos una cuerda al bloque que forme un ángulo θ con la horizontal, la fuerza normal deja de ser igual al peso. La condición de equilibrio en la dirección perpendicular al plano establece la ecuación 5.11.

$$N + F \cdot \text{sen}\,\theta = mg$$ Ecuación 5.11

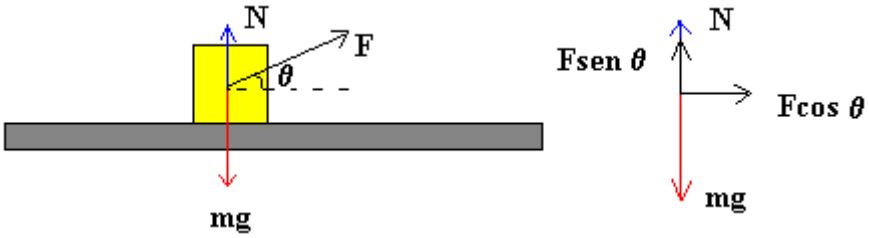


Figura 5.5 Fuerzas que actúan sobre un bloque que se desplaza sobre un plano horizontal.

Veremos ahora la fuerza de rozamiento por deslizamiento. En la figura 5.6, se muestra un bloque arrastrado por una fuerza F horizontal. Sobre el bloque actúan el peso mg, la fuerza normal N que es igual al peso, y la fuerza de rozamiento F_k entre el bloque y el plano sobre el cual desliza. Si el bloque se desliza con velocidad constante la fuerza aplicada F será igual a la fuerza de rozamiento por deslizamiento F_k.

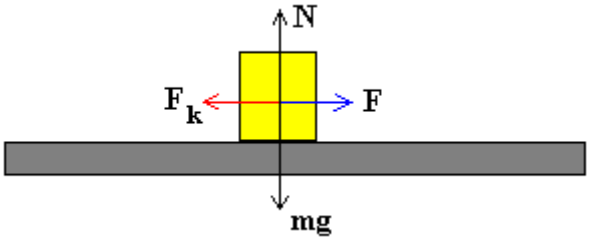


Figura 5.6 Fuerza de rozamiento

Podemos investigar la dependencia de F_k con la fuerza normal N. Veremos que si duplicamos la masa m del bloque que desliza colocando encima de éste otro igual, la fuerza normal N se duplica, la fuerza F con la que tiramos del bloque se duplica y por tanto, F_k se duplica.

La fuerza de rozamiento por deslizamiento F_k es proporcional a la fuerza normal N.

$$F_k=\mu_k N \qquad \text{Ecuación 5.12}$$

La constante de proporcionalidad μ_k es un número sin dimensiones que se denomina coeficiente de rozamiento cinético.
El valor de μ_k es casi independiente del valor de la velocidad para velocidades relativas pequeñas entre las superficies, y decrece lentamente cuando el valor de la velocidad aumenta.
También existe una fuerza de rozamiento entre dos objetos que no están en movimiento relativo.

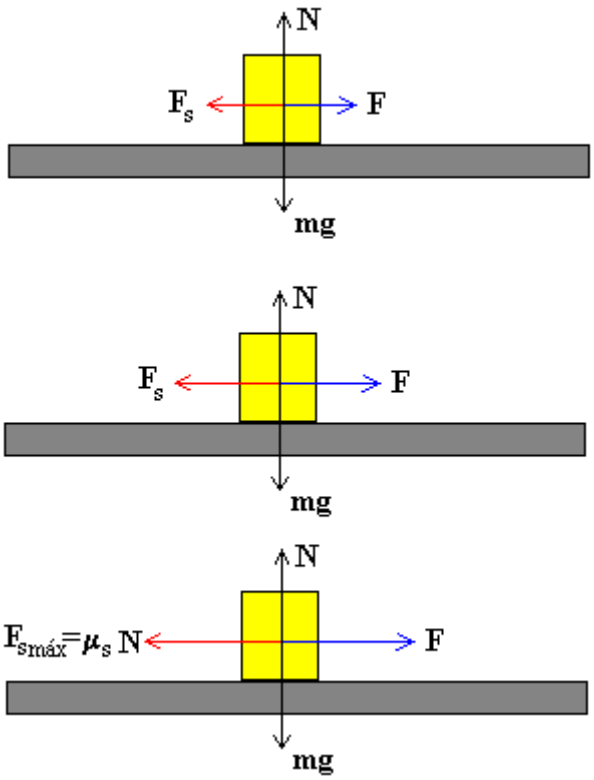


Figura 5.7 Fuerza de rozamiento

Como vemos en la figura, la fuerza F aplicada sobre el bloque aumenta gradualmente, pero el bloque permanece en reposo. Como la aceleración es cero la fuerza aplicada es igual y opuesta a la fuerza de rozamiento F_s.

$$F=F_s \qquad \text{Ecuación 5.13}$$

La máxima fuerza de rozamiento corresponde al instante en el que el bloque está a punto de deslizar.

$$F_{s\,máx}=\mu_s N \qquad \text{Ecuación 5.14}$$

La constante de proporcionalidad μ_s se denomina coeficiente de rozamiento estático.
Los coeficientes estático y cinético dependen de las condiciones de preparación y de la naturaleza de las dos superficies y son casi independientes del área de la superficie de contacto.

5.3.2 Orientación de los objetos

La orientación de los objetos se guarda mediante un quaternión. Vamos a ver primero qué es un quaternión. Para representar una rotación de un ángulo G sobre un eje A (Xa, Ya, Za) el quaternión Q sería el mostrado en la figura 5.8.

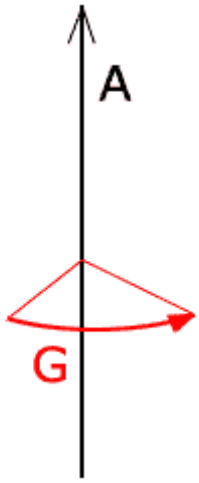


Figura 5.8 Dibujo de un quaternión.

$$Q = (\sin(\frac{G}{2}) \cdot X_a, \sin(\frac{G}{2}) \cdot Y_a, \sin(\frac{G}{2}) \cdot Z_a, \cos(\frac{G}{2})) \qquad \text{Ecuación 5.15}$$

Direct3D (ni otras APIs como OpenGL) no soporta el uso de quaterniones directamente. Para usarlos necesitamos convertir el quaternión en una matriz. Dicha matriz será la matriz que usemos para dibujar el objeto a tratar.

Cuando un objeto con una orientación determinada se posa sobre el terreno, dicho objeto se orienta de tal forma que la normal del objeto esté superpuesta a la normal del terreno en dicho punto.

Para realizar esto primero necesitamos saber la normal del objeto, la obtenemos transformando el vector (0, 1, 0) según la matriz del objeto (según su orientación, su quaternion). Una vez que tenemos la normal del objeto necesitamos la normal del terreno en nuestra posición. Para obtener la normal del plano calculamos los tres puntos que forman el triángulo sobre el que estoy apoyado. Con el producto vectorial de dos de los tres vectores que forman el triángulo obtengo la normal del terreno. Nos aseguramos que la normal tiene la componente “y” positiva.

Una vez que tenemos las dos normales calculamos el ángulo que forman entre ellas por medio de la definición de producto escalar. Así tenemos la ecuación 5.16.

$$\cos(\alpha) = \frac{u \cdot v}{\|u\| \cdot \|v\|} \qquad \text{Ecuación 5.16}$$

Ya tenemos el ángulo que va a formar parte de nuestro quaternión. Ahora nos falta conocer el eje sobre el que giramos. Para obtener el eje de giro hacemos el producto vectorial de las dos normales. Con el eje y el ángulo construimos el nuevo quaternión que representará el giro que tenemos que realizar desde nuestra orientación actual para orientarnos según el suelo.

Para cambiar la orientación que expresa un quaternión respecto a otro quaternión hay que multiplicarlos. Sabiendo esto, multiplicamos el quaternión del objeto por el nuevo. Hay que tener en cuenta que la multiplicación de quterniones no es conmutativa.

Un problema que nos encontramos fue que al principio intentamos calcular la nueva orientación a partir del vector normal (0, 1, 0), es decir, a partir de la normal del objeto en el estado inicial, para así no tener que multiplicar los quaterniones ya que nos daría la orientación final directamente. Al hacer los cálculos a partir de aquí, el objeto se

orientaba bien respecto al plano XY y al plano ZY. Sin embargo, en el plano XZ se orientaba siempre mirando hacia el frente. Por ejemplo, aunque el objeto estuviera rotado mirando hacia la izquierda, al posarse sobre el suelo se ponía mirando hacia el frente. Esto se debía a que el producto vectorial de las dos normales tiene la componente "y" nula o casi nula. De esta forma el quaternión calculado tenía un eje de giro con las coordenadas "y" nulas. Al ser nulas dichas coordenadas, el objeto tenía la orientación inicial, que es mirando hacia el frente.

De la otra forma al multiplicar los quaterniones conservamos la orientación actual del objeto modificada según la normal del terreno.

5.4 COMBUSTIBLE

Para hacer más realista el modelo del objeto a manejar en el simulador le hemos añadido combustible. A la hora de tratar el combustible se tienen en cuenta dos datos, la cantidad de combustible que tiene el objeto y el consumo de combustible máximo que realiza el objeto. Estos dos datos iniciales se leen desde un archivo de configuración por medio de LUA y se utilizan en la función de actualización del objeto. Hay que tener en cuenta que a la masa del objeto hay que añadirle la masa del combustible. Para hacer esto se ha tomado que un litro de combustible equivale a un kilogramo. El consumo se expresa en litros por hora.

Lo primero que se comprueba al actualizar el objeto es si el consumo es mayor que cero, si es así tratamos la cantidad de combustible. Para calcular el consumo que hemos realizado entre el instante anterior y éste usamos la siguiente ecuación que tiene en cuenta la potencia del rotor principal, si la potencia del rotor está al máximo, el consumo será también el máximo. Si el rotor está apagado, no habrá consumo de combustible.

$$Combustibleconsumido = dt \cdot consumoMaximo \cdot \frac{potenciaRotor}{potenciaMaximaRotor}$$

Ecuación 5.17

En la ecuación 5.17 dt es el tiempo transcurrido desde el instante anterior hasta éste.
El valor consumoMaximo será el valor leído del archivo de configuración.
La potencia actual del rotor principal viene dada por potenciaRotor.
Y por último potenciaMaximaRotor es la potencia máxima que puede llegar a tener el rotor principal.

El siguiente paso es comprobar si el combustible consumido en este instante es menor que el combustible que tenemos. Si es menor, restamos al combustible que tenemos el combustible consumido y actualizamos la masa del objeto restándole el combustible consumido. Si el consumo es mayor que el combustible que tenemos, le restamos a la masa del objeto el combustible que tenemos en este instante y ponemos el combustible a cero.

Una vez que tenemos la nueva masa del objeto actualizamos su matriz de inercia. Esto es muy importante porque al disminuir la masa del objeto también disminuye su inercia.

A la hora de actualizar el objeto tenemos en cuenta también que si el combustible es cero los rotores no pueden girar, de esta forma su potencia se va acercando a cero y se estabiliza en el cero. Con esto conseguimos que los rotores no paren de golpe, sino que vayan disminuyendo poco a poco.

6. COLISIONES

En esta sección se explica cómo se realiza la comprobación de colisiones entre los diversos objetos del juego. Se explica también la detección de colisiones mediante esferas y las distintas comprobaciones que se hacen, así como la forma de actuar con unas colisiones u otras.

6.1 COLISIÓN MEDIANTE ESFERAS

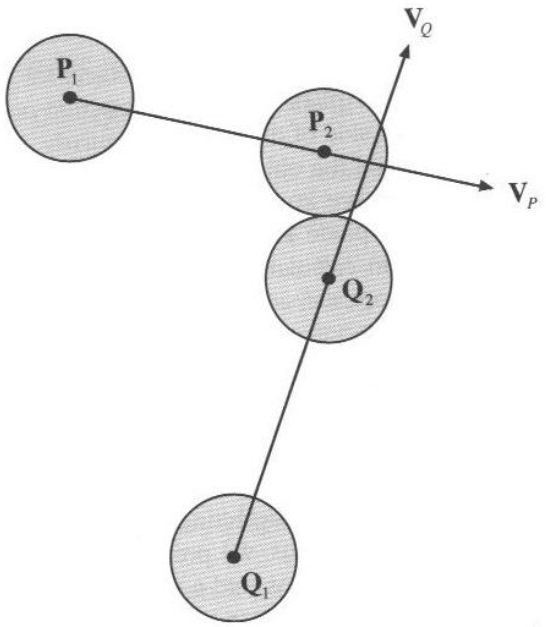


Figura 6.1 Colisión de dos esferas en movimiento

6.1.1 Versión simple

La forma más fácil de hacer una comprobación de colisiones mediante esferas es midiendo la distancia entre los dos objetos en cuestión y comparándola con la suma de los radios. La distancia la calcularemos como distancia al cuadrado para evitar tener que calcular raíces cuadradas que tiene un alto coste. El código sería:

```
BOOL bSphereTest(Objeto3D* obj1, Objeto3D* obj2 )
{
 D3DVECTOR relPos = obj1->prPosition - obj2->prPosition;
 float dist = relPos.x * relPos.x + relPos.y * relPos.y + relPos.z * relPos.z;
 float minDist = obj1->fRadius + obj2->fRadius;
 return dist <= minDist * minDist;
}
```

Código 1

El código 1 tiene un problema, ¿qué pasa si los objetos se están moviendo a una velocidad relativamente grande? En este caso puede ocurrir que al comienzo del frame los objetos no colisionen todavía pero durante su recorrido se crucen y si hubiera colisión. Con este código no detectaríamos dicha colisión y al final del frame aparecerían los objetos como si se hubieran traspasado el uno al otro.

6.1.2 Versión mejorada

Lo que necesitamos es incorporar a nuestro test la velocidad de los objetos de alguna forma.

Sean $\vec{P}_1[t]$ y $\vec{P}_2[t]$ las posiciones de los objetos en el tiempo t. Tomamos $t=0$ como el inicio del frame y $t=1$ como el final del frame. Y sean $\vec{V}_1$ y $\vec{V}_2$ las velocidades de los objetos. Tenemos la ecuación 6.1.

$$\vec{P}_1[t] = \vec{P}_1[0] + t\vec{V}_1$$
$$\vec{P}_2[t] = \vec{P}_2[0] + t\vec{V}_2$$

Ecuación 6.1

Y sus posiciones relativas serán las expresadas en la ecuación 6.2.

$$\Delta\vec{P}[t] = \vec{P}_2[t] - \vec{P}_1[t] = \vec{P}_2[0] + t\vec{V}_2 - \vec{P}_1[0] - t\vec{V}_1 = \left(\vec{P}_2[0] - \vec{P}_1[0]\right) + t\left(\vec{V}_2 - \vec{V}_1\right) = \Delta\vec{P}[0] + t\Delta\vec{V}[0]$$

Ecuación 6.2

La distancia entre los objetos será simplemente la magnitud de la posición relativa, o lo que es lo mismo, la distancia será igual al vector al cuadrado. Siendo * el producto escalar.

$$d^2[t] = \left(\Delta\vec{P}[t]\right)^2 = \left(\Delta\vec{P}[0] + t\Delta\vec{V}[0]\right)^2 = \left(\Delta\vec{P}[0]\right)^2 + 2t\left(\Delta\vec{P}[0] * \Delta\vec{V}[0]\right) + t^2\left(\Delta\vec{V}[0]\right)^2$$

Ecuación 6.3

Ahora todo lo que necesitamos es comprobar si $d[t]$ llega a ser alguna vez menor a la distancia mínima permitida r(la suma de los radios) durante el siguiente frame, que es entre $t=0$ y $t=1$.
El modo de hacerlo es comprobar si la distancia ya es menor que el mínimo(por si acaso no hemos detectado la colisión antes) y si no es así resolvemos la ecuación 6.4.

$$d^2[t] - r^2 = 0$$

Ecuación 6.4

De esta forma calculamos el momento t en el cual los objetos están exactamente a distancia r. Normalmente tendremos dos soluciones, una cuando las esferas se chocan inicialmente y otra cuando las esferas dejan de colisionar entre sí(se traspasan).

$$\Delta\vec{V}^2 t^2 + 2\left(\Delta\vec{P} * \Delta\vec{V}\right)t + \Delta\vec{P}^2 - r^2 = 0$$

Ecuación 6.5

Resolvemos la ecuación 6.6.

$$D = b^2 - 4ac = \left(2\Delta\vec{P} * \Delta\vec{V}\right)^2 - 4\Delta\vec{P}^2\Delta\vec{V}^2 = 4\left(\left(\Delta\vec{P} * \Delta\vec{V}\right)^2 - \left(\Delta\vec{P}^2 - r^2\right)\Delta\vec{V}^2\right)$$

Ecuación 6.6

Si D<0 entonces la ecuación no tiene solución, las esferas no colisionan nunca. Si D>0 entonces las esferas colisionan (Si D=0 entonces las esferas se están tocando). Las raíces de la ecuación serían las que salen de la ecuación 6.7 y 6.8.

$$t_1 = \frac{-2\left(\Delta\vec{P} * \Delta\vec{V}\right) - \sqrt{D}}{2\left(\Delta\vec{V}\right)^2}$$

Ecuación 6.7

$$t_2 = \frac{-2\left(\Delta\vec{P} * \Delta\vec{V}\right) + \sqrt{D}}{2\left(\Delta\vec{V}\right)^2}$$

Ecuación 6.8

Luego podemos coger t_1, comprobar si está entre 0 y 1 y si es así entonces tenemos colisión. Simplificando un poco la ecuación podemos cancelar los 2'es y quedarnos con la ecuación 6.9.

$$t_1 = \frac{-2\left(\Delta\vec{P} * \Delta\vec{V}\right) - \sqrt{D}}{2\left(\Delta\vec{V}\right)^2} = \frac{-\left(\Delta\vec{P} * \Delta\vec{V}\right) - \sqrt{\frac{D}{4}}}{\Delta\vec{V}^2} = \frac{-\left(\Delta\vec{P} * \Delta\vec{V}\right) - \sqrt{\left(\Delta\vec{P} * \Delta\vec{V}\right)^2 - \left(\Delta\vec{P}^2 - r^2\right)\Delta\vec{V}^2}}{\Delta\vec{V}^2}$$

Ecuación 6.9

Otra simplificación más es que podemos calcular si las raíces estarán entre 0 y 1 antes de calcular t_1, de hecho, antes de calcular D. Veamos como hacerlo en la ecuación 6.10.

$$t_1 + t_2 = \frac{-2\left(\Delta\vec{P} * \Delta\vec{V}\right)}{\Delta\vec{V}^2}$$

$$t_1 t_2 = \frac{\Delta\vec{P}^2 - r^2}{\Delta\vec{V}^2}$$

Ecuación 6.10

Este es un caso del llamado teorema de Viete. Miremos los signos de los valores. Si $t_1 t_2 < 0$ entonces podemos decir con seguridad que $t_1 < 0$ y $t_2 > 0$. En ese caso la collision sigue en proceso(se están tocando). Si $t_1 t_2 > 0$ pero $t_1 + t_2 < 0$ entonces $t_1 < 0$ y $t_2 < 0$ y es otro caso fácil: las esferas se están separando una de la otra y no hay colisión.
Para hacerlo más fácil todavía vemos que los denominadores son siempre positivos, así que el signo de las expresiones dependen de los signos de los numeradores. El numerador de la segunda expresión es la distancia entre las esferas en t=0 menos r^2. Esto es lo que calculábamos en el código 1, así que podemos reutilizarlo. Calcular $\Delta\vec{P} * \Delta\vec{V}$ es nuestro siguiente paso.

Podemos encontrar los casos en los que la solución es <0 que es cuando la colisión ya ha tenido lugar. Ahora necesitamos saber los casos en los que la solución es >1. Que es cuando la colisión va a ocurrir pero no en el siguiente frame, así que no nos interesa.

$$t_1 > 1 \Leftrightarrow \frac{-\left(\Delta\vec{P} * \Delta\vec{V}\right) - \sqrt{\left(\Delta\vec{P} * \Delta\vec{V}\right)^2 - \left(\Delta\vec{P}^2 - r^2\right)\Delta\vec{V}^2}}{\Delta\vec{V}^2} > 1$$

Ecuación 6.11

Simplificando llegamos a los dos casos mostrados en las ecuación 6.12.

$$\Delta\vec{V}^2 - (\Delta\vec{P} * \Delta\vec{V}) < 0$$
$$\Delta\vec{V}^2 + 2(\Delta\vec{P} * \Delta\vec{V}) + (\Delta\vec{P}^2 - r^2) > 0$$

Ecuación 6.12

Si los dos casos son ciertos entonces $t_1 > 1$ y sabemos que no hay colisión en el siguiente frame. Ahora bien, si hemos pasado todos los tests no tenemos otra opción que calcular D y ver si es >0. Si queremos saber el tiempo exacto de colisión calcularíamos su raíz cuadrada para hallar t_1. Lo omitimos para simplificar.

```
BOOL bSphereTest(Objeto3D* obj1, Objeto3D* obj2 )
{
        //velocidad □rae□a□e
        D3DVECTOR    dv    = obj2->prVelocity – obj1->prVelocity;
        // Posición □rae□a□e
        D3DVECTOR    dp    = obj2->prPosition – obj1->prPosition;
        //Distancia mínima al cuadrado
        float r = obj1->fRadius + obj2->fRadius;
        //dP^2-r^2
        float pp = dp.x * dp.x + dp.y * dp.y + dp.z * dp.z – r*r;
        //(1)Comprobar si las esferas estaban ya colisionando
        if ( pp < 0 ) return □rae;

        //dP*dV
        float pv    = dp.x * dv.x + dp.y * dv.y + dp.z * dv.z;
        //(2)Comprobar si las esferas se alejan una de la otra
        if ( pv >= 0 ) return false;

        //dV^2
        float vv = dv.x * dv.x + dv.y * dv.y + dv.z * dv.z;
        //(3)Comprobar si hay colisión dentro de 1 frame
        if ( (pv + vv) <= 0 && (vv + 2 * pv + pp) >= 0 ) return false;

        //Discriminante/4
        float D    = pv * pv – pp * vv;
        return ( D > 0 );
}
```

Código 2

6.1.3 Una mejora más

El código 2 tiene un pequeño problema. No es un problema matemático sino un problema relacionado con el ordenador. Si nos fijamos en el último cálculo:

```
float D    = pv * pv - pp * vv;
```

Por ejemplo, sea la distancia entre los objetos 100 unidades y los objetos se mueven a 100 unidades por frame. Entonces pv, vv y pp son normalmente números en torno a 100*100=10000. ¡Luego sus productos serán 10000*10000=100000000! Y tenemos que restar dichos valores. Lo que pasa es que hay una pérdida de precisión donde podemos estar en un error de miles de unidades. Y lo que es peor el signo puede cambiar y no detectar correctamente las colisiones.

Podemos dividir la formula de arriba por vv, eso haría los números más pequeños y más manejables. Como sabemos que vv es mayor que 0 (si fuese 0 pv también lo sería y la comprobación 2 fallaría) entonces el signo no se verá afectado. Gastaremos más ciclos en esta división pero solo llegaremos a este caso cuando todas las comprobaciones anteriores fallen. La última versión del código sería:

```
BOOL bSphereTest(Objeto3D* obj1, Objeto3D* obj2)
{
 //Velocidad relativa
 D3DVECTOR    dv    = obj2->prVelocity - obj1->prVelocity;
 // Posición relativa
 D3DVECTOR    dp    = obj2->prPosition - obj1->prPosition;
 //Mínima distancia al cuadrado
 float r = obj1->fRadius + obj2->fRadius;
 //dP^2-r^2
 float pp = dp.x * dp.x + dp.y * dp.y + dp.z * dp.z - r*r;
 //(1)Comprobar si ya estaban colisionando
 if ( pp < 0 ) return true;

 //dP*dV
 float pv    = dp.x * dv.x + dp.y * dv.y + dp.z * dv.z;
 //(2)Comprobar si se están alejando uno del otro
 if ( pv >= 0 ) return false;

 //dV^2
 float vv = dv.x * dv.x + dv.y * dv.y + dv.z * dv.z;
 //(3)Comprobar si hay colisión dentro de 1 frame
 if ( (pv + vv) <= 0 && (vv + 2 * pv + pp) >= 0 ) return false;

 //tmin = -dP*dV/dV*2
 //El momento en el cual la distancia entre los objetos es mínima
 float tmin = -pv/vv;

 //Discriminante/(4*dV^2) = -(dp^2-r^2+dP*dV*tmin)
 return ( pp + pv * tmin > 0 );
}
```

6.2 COLISIÓN ENTRE OBJETOS DE LA ESCENA

Durante el juego se comprueban cuatro tipos de colisiones, con el terreno, con los objetos estáticos, con los objetos dinámicos y con los proyectiles.

Los proyectiles son tratados como otro tipo de objetos porque están contenidos dentro de los objetos dinámicos, ya que estos últimos son los que tienen la habilidad de disparar.

6.2.1 Colisión con el terreno

Para detectar la colisión con el terreno lo hacemos la forma más simple posible. Comprobamos que la posición del objeto menos el radio de colisión es mayor

que la altura del terreno en dicho punto. Si no es así hay colisión y actualizamos la posición del objeto según la altura del terreno.

Si la colisión se produce a una velocidad mayor de la determinada se activará en el objeto un sistema de partículas simulando fuego y además se activará una explosión en el objeto.

6.2.2 Colisión con objetos estáticos

En una primera versión teníamos una lista con todos los objetos estáticos del escenario. Recorríamos toda la lista e íbamos comprobando si había colisión o no. Funcionaba pero había un gran problema, si había 1000 objetos recorríamos a las malas los 1000 objetos. No era eficiente.

Como mejora en vez de una lista de objetos tenemos una lista de secciones de terreno. El terreno se divide en porciones, una cuadrícula exactamente. Cada sección del terreno contiene una lista de objetos. Para comprobar las colisiones lo que hacemos es acceder a la lista de objetos de la sección en la que estamos y recorrerla para hacer la comprobación. Esta forma nos reduce considerablemente el número de comprobaciones que realizamos.

Si hay colisión hacemos que los objetos reboten según la velocidad que llevaban. El rebote se hace para que no se bloqueen los objetos en la detección de la colisión. Por ejemplo pensemos en una colisión que queremos que deje la velocidad de los objetos a 0. Si hay colisión pondríamos las velocidades a 0 pero si no actualizamos su posición, aunque queramos ir en otra dirección en la cual no haya colisión como estamos ya colisionando siempre entraremos en el caso de que hay colisión y de establecer la velocidad a 0. Si modificamos la posición con un rebote, podremos cambiar de dirección y seguir avanzando.

Si hay colisión ambos objetos activarán un sistema de partículas propio simulando fuego.

6.2.3 Colisión con objetos dinámicos y proyectiles

Con los objetos dinámicos y los proyectiles recorremos una lista de objetos para ir comprobando si hay colisión o no.

Intentamos hacer la misma operación que con los estáticos pero había que modificar muchas veces las secciones del terreno y logramos mejores resultados de esta manera. Siempre y cuando no haya demasiados objetos dinámicos en la escena, ya que recorremos en el caso peor toda la lista de objetos dinámicos. No se puede hacer como en los objetos estáticos ya que no podemos asegurar si los objetos susceptibles de colisión son los que se encuentran en la misma región que el helicóptero. Ya que al tener velocidad puede que en un instante estén en otra región pero con una velocidad suficientemente alta como para colisionar en el siguiente paso.

Si hay colisión con los objetos dinámicos se actúa igual que en el caso de los objetos estáticos.

Si es con un proyectil, no hay rebote y en vez de activar el sistema de partículas se activa una explosión en el objeto en cuestión.

7. VENTANA PRINCIPAL

En esta sección se explican las mejoras realizadas en la ventana del simulador. Así como el uso de mensajes, recursos y demás opciones de las ventanas en Windows.

7.1 SISTEMA DE MENSAJES

7.1.1 Captura de mensajes

La idea principal de la ventana es que nunca nos salgamos de ella, ya que si lo hacemos, Windows la destruirá inmediatamente. Así que tenemos que hacer un bucle principal que siempre se ejecute, a no ser que queramos salir expresamente de él.

Windows genera eventos llamados *Windows Messages* cuando ocurren ciertas cosas. Si intentamos cerrar la ventana con el botón X, Windows genera un mensaje de cierre y lo pone en una cola de mensajes que tiene la ventana.

Lo único que tenemos que hacer es comprobar dicha cola de mensajes para saber qué está pasando y cómo actuar. Para que el programa mire lo que hay en la cola necesitamos generar un código especial que siga la estructura de la figura 7.1.

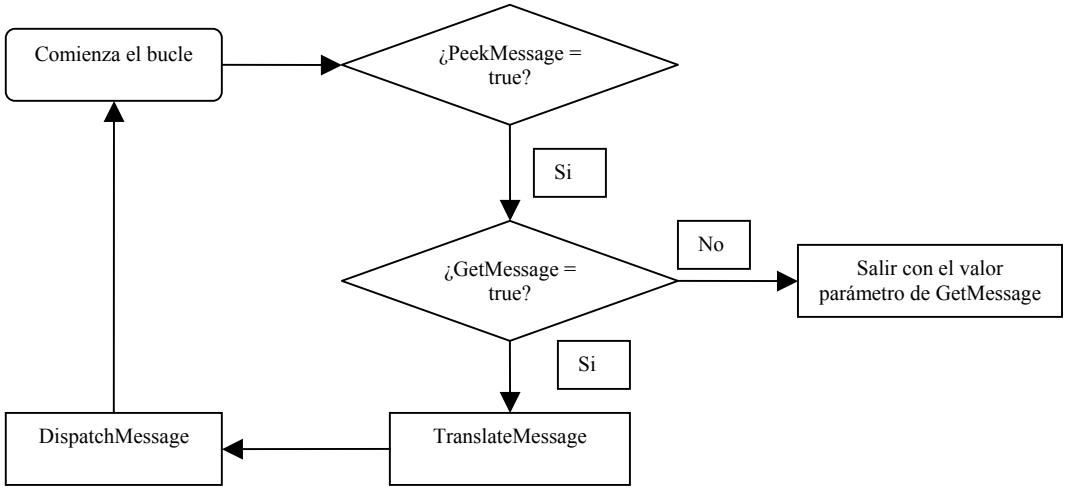


Figura 7.1 Estructura del tratamiento de mensajes de una ventana.

Como se puede ver en la figura 7.1 la única forma de salir del bucle es con la llamada a GetMessage.

7.1.2 Mensajes

Todos los mensajes de Windows tienen un tipo que comienza con WM_. Por ejemplo, si presionamos una tecla se emitirá un mensaje de WM_KEYDOWN. Todos los mensajes tienen la misma estructura:

```
Typedef struct tagMSG{
        HWND hwnd;//objetivo del mensaje
        UINT message; //tipo de mensaje
        WPARAM wParam; //parámetro
        LPARAM lParam; //parámetro
        DWORD time; //Momento cuando se recibió el mensaje
        POINT pt; //posición del cursor cuando se generó el mensaje}
```

Se puede ver en la estructura que los mensajes pueden tener parámetros, por ejemplo, un mensaje WM_KEYDOWN tiene unos parámetros específicos para saber qué tecla se ha pulsado.

El parámetro hwnd indica a qué ventana iba dirigido el mensaje.

7.1.3 PeekMessage

Esta función se puede usar de varias maneras, para ver rápidamente si hay mensajes esperando, para coger un determinado mensaje o para filtrar mensajes.

Devuelve cierto si hay algún mensaje esperando y falso si no lo hay.

7.1.4 GetMessage

Si no hay mensajes en la cola esta función espera hasta que los haya. Debido a esto esta función se llama cuando sabemos con seguridad que hay mensajes (PeekMessage). GetMessage devuelve 0 si recibe un mensaje de cierre de programa.

7.1.5 TranslateMessage

Recibe un puntero a un mensaje y devuelve cierto o falso. Lo que esta función hace es eliminar mensajes redundantes de la cola de mensajes. Pos ejemplo, si tenemos en la cola dos mensajes de tipo WM_KEYDOWN y WM_KEYUP los dos mensajes nos dicen lo mismo, que se ha pulsado una tecla. Esta función deja sólo uno de los dos mensajes en la cola. Cuando hemos traducido el mensaje damos vía libre.

7.1.6 DispatchMessage

Esta función recibe un mensaje, mira a quién iba dirigido el mensaje y llama al método WndProc() de la ventana que ya se encarga de actuar según el mensaje. Es decir, lo que hace es repartir los mensajes a las ventanas apropiadas.

7.2 WNDPROC()

Este método se encarga de realizar las acciones que le indican los mensajes recibidos.

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam) ;

Empezaremos con el valor de retorno. LRESULT es realmente un long. El convenio general es que sea distinto de cero si se ha realizado la operación con éxito y cero si no es así.

CALLBACK le indica a Windows que es una función especial, es un requerimiento que necesita Windows para llamar correctamente al método.

Veamos los parámetros que recibe el método:

hwnd. Es la ventana. Este método está asociado a una clase de ventana, no a una ventana en concreto. Es una forma de poder identificar distintas ventanas de la misma clase. Hwnd se refiere a la ventana que recibió el mensaje.

iMsg. Este es el identificador del mensaje. Puede ser WM_MOUSEMOVE, WM_KEYDOWN, etc.

wParam y lParam. Son los argumentos del mensaje. Dependen del tipo de mensaje que se recibió.

Dentro del método lo primero que tenemos que hacer es una distinción de casos dependiendo del tipo de mensaje:

```
switch(iMsg) {
        case WM_CREATE:
        break;
        case WM_DESTROY:
        break;
        case WM_ACTIVATEAPP:
        break;
        case WM_KEYDOWN:
        break;
        …
}
```

En algunos casos tendremos que hacer otra distinción de casos dependiendo del parámetro que recibamos. Por ejemplo si recibimos un mensaje de comando:

```
case WM_COMMAND:
        switch( LOWORD(wParam) )
        {
                case ID_VISTAS_SEGUIMIENTO:
                        camera = (Camera *)&followCamera;
                 break;
                case ID_SALIR:
                        PostQuitMessage(0);
                        break;
                break;
                …
        }
        break;
```

Tenemos que distinguir los distintos comandos, en este ejemplo se han usado identificadores de recursos que indican la pulsación de un botón.

7.3 RECURSOS

Básicamente un recurso es algo que nuestro programa usa para ejecutarse y no es parte del código. Para los juegos, los recursos pueden ser imágenes, sonidos, etc. Para aplicaciones Windows los recursos también incluyen iconos, cursores, ventanas de diálogo y menús. Normalmente también se tienen recursos específicos para cada aplicación, como URLs, etc.

Windows tiene un conjunto de funciones API que podemos usar para manejar los recursos y guardarlos en nuestro archivo ejecutable.

Esto es bueno por dos razones. La primera es que mantiene todas las cosas en un mismo sitio y la segunda que hace más difícil a los demás cambiar cosas que no deberían.

Cada recurso tiene un identificador que puede ser o bien un string o bien un entero (normalmente es un entero). Usamos dicho identificador para indicarle a Windows qué recurso queremos que use. Los identificadores son únicos dentro de un mismo programa, no son globales. De esta forma diferentes programas pueden usar los mismos identificadores.

Para hacer más manejable la utilización de recursos y no manejar enteros, creamos una clase *resource.h* la cual define constantes con el valor de los enteros identificadores de recursos.

Algunos de los tipos básicos de recursos son:

- Diálogos y menús. Las cajas de diálogo y los menús se salvan en el script de recursos como texto. Este texto le dice a Windows qué controles de diálogo hay que usar y dónde. Más o menos es lo mismo que hacen las páginas web que se salvan como HTML. No guardas el dibujo de la página, guardas una descripción de ella y dejas al explorador que recree la página. Aquí no se guarda el dibujo del diálogo, se guarda una descripción y se deja a Windows que lo recree.
- Bitmaps, iconos y cursores. Estos recursos se guardan en archivos separados del script de recursos (es similar a la etiqueta <IMG> en HTML).
- Tablas de strings. Sirve por ejemplo para guardar todos los mensajes de error. Así si los tienes que usar los cargas y ya está, no tienen que permanecer en memoria.
- Teclas rápidas. Le dicen a Windows cómo ha de actuar si se presionan ciertas combinaciones de teclas (por ejemplo, en un editor de texto Ctrl.+C sería el comando de copiar).

- Todos los demás recursos se consideran recursos personalizados. Esto quiere decir que lo único que va a hacer Windows es comprimirlo en el archivo EXE y cuando los tenga que utilizar sacarlos del archivo EXE.

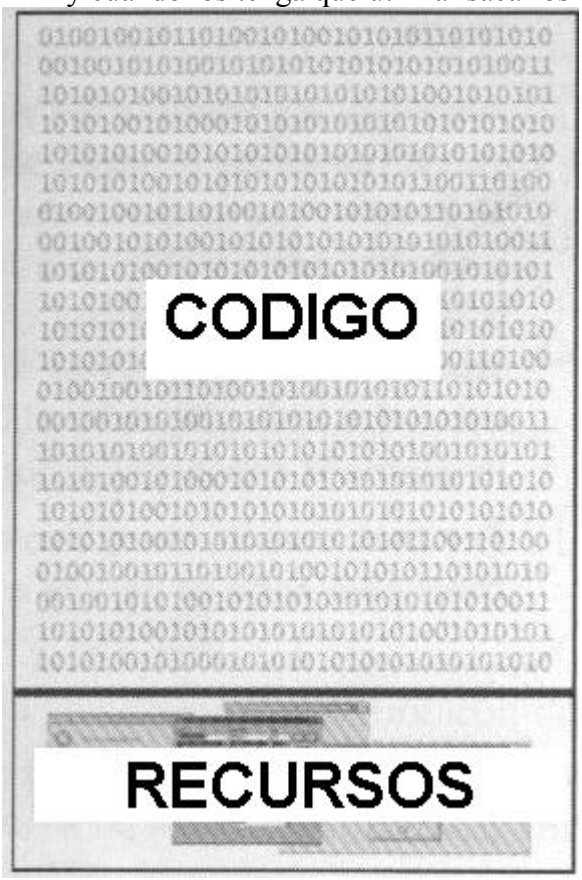


Figura7.2 Esquema de un archivo .EXE

7.3.1 Iconos

Para aplicar un icono que aparezca en el explorador de Windows necesitamos crear primero un icono en los recursos de la aplicación. Se necesitan dos tipos de icono según su tamaño, un icono pequeño de 16x16 pixels y un icono grande de 32x32 pixels.

Una vez creados los iconos hay que darles un nombre a sus identificadores. Supongamos que les llamamos IDI_ICON_PEQ y IDI_ICON_GRA. Para que el icono aparezca en el explorador hay que establecer el icono en la ventana. Para ello cargamos los iconos y se los asignamos a las propiedades HICONs de la ventana cuando la vayamos a registrar.

```
//Establecer y registrar la ventana.
WNDCLASSEX wc;
…
wc.hIcon=LoadIcon(hInstance, MAKEINTRESOURCE(IDI_ICON_GRA));
wc.hIconSm=LoadIcon(hInstance, MAKEINTRESOURCE(IDI_ICON_PEQ));
…
RegisterClassEx(&wc);

//Crear ventana
…
```

En el ejemplo hemos usado MAKEINTRESOURCE. Esta macro está definida en Windows y toma un entero como parámetro y lo convierte en un valor compatible con las funciones que manejan recursos.

7.3.2 Cursores para el ratón

Para embellecer un poco la aplicación se puede cambiar el cursor predeterminado de Windows cuando esté sobre la ventana de la aplicación. Para ello actuamos de una forma similar a los iconos, si suponemos que tenemos el cursor creado en los recursos con el identificador IDC_PUNTERO escribiríamos:

```
WNDCLASSEX wc;
…
wc.hCursor=LoadCursor(Inst., MAKEINTRESOURCE(IDC_PUNTERO));
…
```

Con este fragmento de código conseguimos que el cursor sea siempre el que hemos cargado. Si el cursor tiene que cambiar por alguna razón, habrá que captar el mensaje WM_SETCURSOR. Lo único que tenemos que hacer es capturar dichos mensajes con el método *WndProc()* y elegir qué cursor es el que debemos establecer usando el método *SetCursor*.

7.3.3 Menú de opciones

La barra de menú de la ventana también ha de crearse como un recurso. Su funcionamiento consiste en asignarle un identificador a cada botón del menú que se pueda ejecutar, es decir, excluyendo los botones desplegables de menús. Cuando se presiona un botón del menú Windows le envía un mensaje a la ventana de tipo WM_COMMAND con el identificador de recurso del botón. Básicamente lo que le indica Windows a la ventana es que debe ejecutar el comando asociado al identificador.

En nuestra aplicación captaríamos el mensaje con el método *WndProc()* y dependiendo del identificador actuaríamos de una forma u otra.

8. DIRECTX

En esta sección se habla del manejo de dos partes de DirectX. Nos centramos en el control de un joystick por medio de DirectInput y en la reproducción de un video con la ayuda de DirectShow y DirectSound. Como última sección explicamos el uso de mapas de acciones para leer los dispositivos con DirectInput.

8.1 IDEAS GENERALES

DirectX está implementado usando COM (Microsoft's Component Object Model). A todas las cosas de DirectX se accede a través de una interfaz COM.

Podemos obtener dichas interfaces llamando a funciones de DirectX. La única regla básica a tener en cuenta trabajando con interfaces es que hay que borrar siempre lo que se crea. Si no borramos las interfaces que no usemos estamos perdiendo memoria de la tarjeta gráfica o de la tarjeta de sonido y puede provocar que el juego se nos pare en poco tiempo. Para borrar las interfaces hay que llamar a la función Release():

```
m_iDirectGraphics->Release()
```

Para que nuestra aplicación pueda usar objetos COM hay que llamar a una función de inicialización: CoInitialize(). Para terminar hay que llamar a la función CoUninitialize().

Para crear objetos COM hay que llamar a la función CoCreateInstance(). Aunque no siempre es así, algunos componentes de DirectX tienen su propia función de creación, por ejemplo en DirectInput se llamaría a DirectInput8Create().

8.2 DIRECTINPUT

DirectInput proporciona a nuestro juego una interfaz para el teclado, ratón y joystick. Aunque Windows ya tiene sus propias interfaces para leer desde teclado, ratón o joystick. Sin embargo DirectInput es mucho más flexible porque se sitúa encima de todos los demás drivers.

Para hacer cualquier cosa con DirectInput primero hay que crear una interfaz con DirectInput8Create(). Una vez creado le pedimos a DirectInput que nos diga qué dispositivos hay disponibles para leer (teclados, ratones o joysticks). Cada dispositivo tendrá un identificador. Podemos obtener el control de algún dispositivo llamando a la función CreateDevice() para el identificador del dispositivo.

Para saber qué tipo de controles tiene el dispositivo (eje X, eje Y, rotación Z, teclas, etc.) hay que llamar a la función EnumObjects(), cada control tendrá un identificador. Podemos establecer los valores adecuados para cada control, por ejemplo, dar unas coordenadas máximas y mínimas del eje X de un joystick.

Para leer cualquier control de un dispositivo tenemos que llamar primero a la función Acquired(), en el caso de que no tengamos adquirido el dispositivo. Cuando adquirimos un dispositivo ganamos acceso a su lectura.

8.2.1 Joystick

Para comprobar si hay algún joystick conectado ejecutamos el siguiente comando:

```
lpDI->EnumDevices( DI8DEVCLASS_GAMECTRL,
                   EnumJoysticksCallback,NULL,
                   DIEDFL_ATTACHEDONLY );
```

Cómo parámetro le pasamos el nombre del método EnumJoysticksCallback, este último método lo que hace es crear un nuevo objeto con CreateDevice() para poder tener acceso al joystick, en caso de que exista alguno.

El siguiente paso es especificar un formato de datos para ese joystick. Este formato indicará en qué controles estamos interesados y cómo deben ser tratados. Tenemos que pasarle una estructura DIJOYSTATE2 a IDirectInputDevice::GetDeviceState().

```
g_pJoystick->SetDataFormat( &c_dfDIJoystick2 );
```

Lo siguiente es especificar el nivel de cooperación con otros dispositivos de DirectInput, para saber cómo ha de interactuar con el sistema y con ellos.

```
g_pJoystick->SetCooperativeLevel( hwnd, DISCL_EXCLUSIVE |
                                  DISCL_FOREGROUND );
```

Por último hay que enumerar los controles del joystick, la función callback permite establecer propiedades de los controles como los valores mínimo y máximo de los ejes encontrados.

```
g_pJoystick->EnumObjects( EnumObjectsCallback,
                          (VOID*)hwnd, DIDFT_ALL );
```

Para configurar los ejes usamos la estructura DIPROPRANGE, establecemos sus valores y se la asignamos al joystick:

```
g_pJoystick->SetProperty( DIPROP_RANGE, &diprg.diph )
```

En el código del proyecto está la clase InputManager que se encarga de todo lo relacionado con DirectInput. A parte del joystick, controla también el teclado y el ratón, en caso que haya dichos dispositivos.

8.3 DIRECTSHOW

Este componente de DirectX facilita la reproducción de videos y audio de alta calidad entre otras cosas.

DirectShow se basa en filtros para trabajar. Un filtro de DirectShow coge un archivo, realiza una operación en particular sobre el archivo y devuelve los resultados. Por ejemplo, un filtro puede recibir un archivo en MPEG, descomprimirlo, y devolver el video descomprimido.

DirectShow trabaja encadenando varios filtros, la salida de uno es la entrada de otro y así sucesivamente. Esta cadena es lo que forma el denominado grafo de filtros, que puede ir desde un archivo en memoria hasta el video mostrado por pantalla.

DirectShow también contiene un componente que maneja grafos de filtros. Por ejemplo, le podemos decir al manejador ¡ejecútate! Y él se encarga de todos los detalles de obtener los datos apropiados y que dichos datos sigan el flujo de filtros correcto. Este manejador de grafos nos sirve también para construir grafos automáticamente que muestran un determinado video por pantalla.

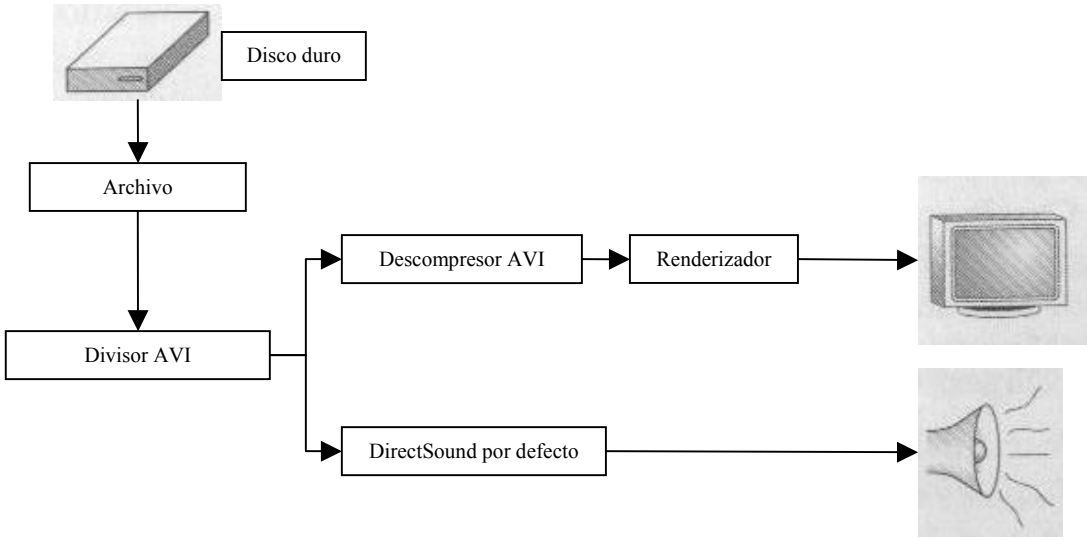


Figura 8.1 Esuqema del funcionamiento de DirectShow

Para mostrar un video primero tenemos que inicializar la aplicación para usar objetos COM llamando a Coinitialize (). Lo siguiente en hacer es crear una instancia del manejador de grafos:

```
CoCreateInstance(CLSID_FilterGraph, NULL,
 CLSCTX_INPROC_SERVER,IID_IGraphBuilder, (void **)&pGraph);
```

Con el manejador creamos el resto de objetos necesarios:

```
pGraph->QueryInterface(IID_IMediaControl, (void **)&pMediaControl);
pGraph->QueryInterface(IID_IMediaEvent, (void **)&pEvent);
pGraph->QueryInterface(IID_IVideoWindow, (void **)&pVidWin);
pGraph->QueryInterface(IID_IMediaSeeking, (void **)&pMediaSeeking);
```

IMediaControl es la interfaz encargada de empezar a reproducir el video.
IMediaEvent sirve para saber cuándo ha acabado el video.
IVideoWindow es la ventana en la cual se reproducirá el video.
IMediaSeeking es para establecer el comienzo del video en un determinado punto del video.

```
pGraph->RenderFile(temp, NULL);
pVidWin->put_Owner((OAHWND)&main_win);
pVidWin->put_WindowStyle(WS_CHILD | WS_CLIPSIBLINGS);
pVidWin->SetWindowPosition( 0,0,GetSystemMetrics(SM_CXSCREEN),
              GetSystemMetrics(SM_CYSCREEN));
```

Con la llamada a RenderFile establecemos el grafo necesario para decodificar el archivo dado en la variable temp y mostrarlo por pantalla. Los métodos put_Owner y put_WindowStyle establecen el padre de la ventana donde se mostrará el video y el estilo de dicha ventana. El método SetWindowPosition establece la posición de la ventana de reproducción del video, en este caso está puesto en modo ventana completa.

```
LONGLONG pos = 0;
pMediaSeeking->SetPositions(&pos,
        AM_SEEKING_AbsolutePositioning,NULL,
        AM_SEEKING_NoPositioning);
```

Con SetPosition establecemos el comienzo del video en la posición 0, al principio.

```
while(pEvent->WaitForCompletion(0, &evCode)!=VFW_E_WRONG_STATE)
{
        if(K_ENTER)
                break;
}
pMediaControl->Stop();
```

Mientras no detectemos el final del video, o el usuario presione la tecla "enter", esperamos. Cuando finalice paramos el video con Stop. No hay que olvidarse al final de eliminar las interfaces que no vayamos a usar más.

Con esto logramos mostrar un video por pantalla. En nuestro proyecto hemos usado la clase CMovie que se encarga de realizar todas estas operaciones. Lo usamos para mostrar un video a modo de introducción en nuestro proyecto.

8.4 MEJORA DEL TRATAMIENTO DE DISPOSITIVOS USANDO DIRECTINPUT

Una de las novedades de directX es el mapeo de acciones. Este método nos ayuda a escapar de los detalles de definir a bajo nivel botones, teclas y ejes para poder centrarnos en los controles que necesita el juego. Por ejemplo, usando acciones le podemos decir a DirectInput qué se puede hacer en el juego (moverse, dispara, usar objeto, etc.), y DirectInput se encargará de los detalles de mapear dichas acciones del juego con las teclas, botones, etc.

Usando acciones le podemos dar a DirectInput información adicional sobre nuestro juego para que asigne los controles apropiados a nuestras acciones. Por ejemplo, podemos indicarle a DirectInput que nuestro juego es de carreras, y DirectInput automáticamente sabe que vamos a preferir usar un joystick de volante. De esta forma hay unos controles predefinidos para cada tipo de juego.

8.4.1 Resumen del uso de los mapas de acciones

1. Establecer nuestro mapa de acciones. El conjunto de acciones que necesitan ser mapeadas a dispositivos.
2. Especificar el género del juego (carreras, naves espaciales, arcade, etc..).
3. asignar las acciones de nuestro mapa de acciones a controles por defecto.
4. Enumerar los dispositivos que mejor se ajustan al mapa de acciones que hemos creado.
5. Para cada jugador, seleccionar un dispositivo y configurarlo según el mapa de acciones.
6. Leer los datos de los dispositivos.

8.4.2 Identificadores del mapa de acciones

Un mapa de acciones no es más que una lista de identificadores que el programa usa para distinguir las acciones:

```
#define INPUT_ROTORAON 3L
#define INPUT_ROTORAOFF 4L
```

Estos identificadores serán devueltos por DirectInput a nuestro programa cuando ocurren los eventos. DirectInput nos envía un mensaje cuando ocurre una acción. De hecho, el mapa de acciones le indica a DirectInput qué mensaje nos gustaría que fuera. Tendremos que hacer una distinción de casos para saber qué acción ha de realizarse.

Hay dos tipos de acciones, sobre botones y sobre ejes. A veces, una misma acción puede realizarse mediante los dos tipos de acciones. Por ejemplo, girar a la izquierda se puede hacer con el eje x del joystick o también con la flecha izquierda del teclado. Tenemos que tener en cuenta esto y poder manejar las dos situaciones.

Una acción de un eje del joystick se deberá leer a partir de la posición del eje asociado a la acción (dada por DirectInput), y a partir de ahí, tenemos que determinar cuando el jugador está girando a la izquierda o a la derecha. DirectInput nos envía un mensaje que nos dice: "Giro: El eje está en la posición -589" y sabemos por el mensaje que el jugador está torciendo el joystick hacia la izquierda. Para una acción de giro necesitaremos tres acciones, una asignada a un eje, otra asignada a un botón para ir hacia un lado y otra acción para ir hacia el otro lado. Las teclas y los botones se detectan al ser pulsados, deberemos leer true o false.

8.4.3 Asignación de acciones

Para asignar acciones a determinados botones, teclas o ejes debemos usar la estructura DIACTION definida en DirectInput. Esta estructura consta de una lista de elementos con ocho atributos, algunos de ellos opcionales:

uAppData. Identificador de la acción.

dwSemantic. El control al que queremos asignar la acción.

dwFlags. Información adicional. Por ejemplo, si ponemos DIA_APPFIXED el jugador no podrá cambiar la configuración de esta acción.

lptszActionName, uResIdString. Nombre de la acción o un identificador de recurso para mostrar por pantalla.

guidInstance. Para asignar esta acción a un dispositivo concreto, Normalmente GUID_NULL.

dwObjID. El identificador del control. Normalmente 0.

dwHow. Es 0.

Un ejemplo de DIACTION sería:

```
DIACTION g_rgGameAction[NUMBER_OF_SEMANTICS] =
{
   //Mapa del teclado
   { INPUT_LUZ1,       DIKEYBOARD_NUMPAD2,    0, _T("Luz delantera"), },
   { INPUT_ROTORAON,    DIKEYBOARD_1,       0, _T("Encender motor"), },
   { INPUT_ROTORAOFF,   DIKEYBOARD_0,       0, _T("Apagar motor"), },
   { INPUT_SALIR,       DIKEYBOARD_ESCAPE,  DIA_APPFIXED, _T("Salir"), },
         .
         .
         .
};
```

8.4.4 Asignación del mapa de acciones

Para asignar el mapa de acciones a los dispositivos hay que detectarlos y enumerarlos. Para ello hay que llamar al método EnumDevicesBySemantics del interfaz IDirectInput8. Esta función hace uso de un método de tipo CALLBACK. Este último método debe realizar lo siguiente:

Construir el mapa de acciones. Debe tomar el mapa de acciones actual y el dispositivo a tratar y pedirle a DirectInput que construya un mapa de acciones para el dispositivo. Hacemos esto llamando al método BuildActionMap de IDirectInputDevice9. Es como decirle a DirectInput: "Si tuvieras que asignar estas acciones a este dispositivo,¿cómo lo harías y por qué?".

Mirar el mapa de acciones que ha construido DirectInput para saber cómo lo ha construido y porqué.

Sobrescribir el mapa de acciones si es necesario. Normalmente no vamos a querer sobrescribir el mapa de acciones que ha construido DirectInput pero podemos hacerlo.

Asignar el mapa de acciones al dispositivo. Cuando todo está como nosotros queremos asignamos el mapa al dispositivo. Esto crea una unión entre las acciones y los controles físicos. Hay que llamar al método SetActionMap.

8.4.5 Lectura de datos de un dispositivo

Para leer los datos de un dispositivo hay que adquirir las acciones del dispositivo llamando al método GetDeviceData(sizeof(DIDEVICEOBJECTDATA), adod, &dwItems, 0);. El parámetro adod es un DIDEVICEOBJECTDATA y dwItems es un DWORD. El número de eventos de entrada estará almacenado en dwItems, y todos los eventos estarán almacenados en el array "adod". Cada evento tiene su tipo almacenado en adod.uAppData, y el dato actual almacenado en adod.dwData. Los datos que no se refieren a ejes se reciben como "button pressed" o "button released". Hay que tener en cuenta esto para actuar de forma adecuada. Una vez que tenemos adod y dwItems iteramos con el número de eventos. Para cada evento hacemos una distinción de casos según adod.uAppData. Este atributo tendrá el identificador de la acción, así que es fácil distinguir los casos. Los resultados se guardarán en una estructura de tipo PLAYERDATA que tendrá los valores apropiados, trae o false para los botones y un DWORD para cada eje. Esta estructura se pasa a los diferentes objetos que necesitan actualizarse mediante los eventos de entrada.

8.4.6 Configuración de los dispositivos

Para configurar los controles basta con llamar al método ConfigureDevices de IDirectInput8. Después de llamar a este método DirectInput entiende que los dispositivos ya no tienen la configuración adecuada y hay que realizar los cambios adecuados según el usuario.

8.5 ARCHIVOS .X

Los objetos tridimensionales se simulan en un ordenador mediante una malla poligonal que represente aproximadamente al objeto que queremos mostrar.

Una malla es un conjunto de puntos en el espacio y de aristas que los conectan. Los puntos y las aristas forman caras, también llamadas polígonos; es al representar estas caras con un relleno cuando la malla tiene un aspecto de objeto sólido. No entraremos en detalles sobre las propiedades que debe cumplir una malla para que defina un objeto representable.

La siguiente cuestión a abordar es la construcción de una malla tridimensional para representar el objeto que queremos en nuestra aplicación. La forma más simple es definir directamente los vértices y las aristas a mano, pero ésta solución en la práctica resulta inaplicable al intentar definir objetos tridimensionales con un mínimo de complicación, ya que a la hora de dar apariencia "real" a estor objetos entran en juego consideraciones más complejas que las aristas y los vértices, como las normales de una cara o un vértice, los materiales asignados a la malla o las texturas, iluminaciones…etc.

Todas estas consideraciones, aunque se podrían intentar hacer a mano, conllevan un trabajo abrumador, tanto de cálculo previo cómo de tiempo de computación a la hora de ejecutar el programa.

Por eso la creación de elementos tridimensionales complejos se realiza de manera externa a la aplicación, hay multitud de programas especializados en la creación de figuras tridimensionales como pueden ser Maya, 3D Studio, o Blender, por citar algunos. Es esta diversidad de orígenes lo que conlleva que haya multitud de formatos para representar un objeto tridimensional y, aunque los fundamentos básicos son los mismos, hay diferencias a la hora de crear un archivo con uno u otro. Sin embargo son éstas bases comunes las que hacen que sea posible la conversión de un formato a otro. DirectX tiene un formato de archivos propio, el formato .x, y es el que nosotros usamos en nuestra aplicación.

8.5.1 Formato de los archivos .X.

A partir de información encontrada en las páginas [15] y [16], intentaremos describir cómo es la estructura básica de un archivo con formato .x

El formato del archivo de DirectX proporciona un formato plantilla-conducido rico del archivo que permita el almacenado de mallas, texturas, animaciones y objetos definibles por el usuario. El soporte para los sistemas de animación permite que las trayectorias predefinidas sean almacenadas para reproducirse en tiempo real. También da soporte para instanciamientos múltiples a un objeto, tal como una malla, mientras que almacena sus datos solamente una vez por archivo, y jerarquías para expresar relaciones entre los registros de datos. El formato del archivo de DirectX es utilizado de forma nativa por el API Direct3D, proporcionando soporte para la lectura de objetos predefinidos en una aplicación o la escritura de información sobre la construcción de una malla por una aplicación en tiempo real.

Éste formato de archivos proporciona primitivas de datos de bajo nivel, sobre los cuales las aplicaciones definen primitivas de un nivel más alto a través de plantillas. Éste es el método por el cual Direct3D define primitivas de más alto nivel como vectores, mallas, colores y matrices.

El formato de archivo de directX es un formato independiente de la arquitectura y del contexto. Está basado en plantillas y puede ser usado por cualquier aplicación cliente, actualmente es usado por las librerías de Direct3D para describir datos de geometría, jerarquías de marco y animaciones.

Cabecera:
La cabecera del fichero es opcional, pero debe aparecer al principio de la secuencia de datos. La cabecera contiene lo siguiente:

Tipo	*Sub tipo*	*Tamaño*	*Contenido*	*Significado del contenido*
Número mágico – requerido		4 bytes	`"xof "`	
Número de versión – requerido	Número principal Número secundario	2 bytes 2 bytes	`03` `02`	Versión Principal 3 Versión Secundaria 2

Tipo del formato – requerido		4 bytes	`"txt "`
			`"bin "`
			`"com "`
				Archivo de texto
				Archivo binario
				Archivo comprimido

Tipo de compresión – requerido si el tipo de formato es comprimido		4 bytes	`"lzw "`
			`"zip "`
			`etc…`	

Tamaño de los float – requerido		4 bytes	`0064` `0032`	Floats de 64 bits Floats de 32 bit

Ejemplo-- xof 0302txt 0064

Comentarios:
Los comentarios sólo son aplicables en archivos de texto. Pueden aparecer en cualquier parte de la secuencia de datos. Un comentario empieza o con la doble barra del estilo de C++ (//) o con el carácter almohadilla (#). El comentario seguirá hasta la fin de línea.

Plantillas:
Las plantillas definen cómo interpretar la secuencia de datos. Los datos son separados en módulos por la definición de la plantilla. Una plantilla tiene la siguiente forma:

template <nombre_plantilla> {
<UUID>
<miembro 1>;
…
<miembro n>;
[restricciones]
}

nombre_plantilla: Es un nombre alfanumérico que puede contener el carácter subrayado (_) y no debe empezar por un número.

UUID: Un identificador único universal formateado según el estándar OSF DCE rodeado por paréntesis angulares(< y >). Por ejemplo: <3D82AB43-62DA-11cf-AB39-0020AF71E433>

Miembros: Los miembros de la plantilla consisten en un tipo de datos declarado seguido por un nombre opcional o un array de un tipo de datos declarado. Los tipos de datos primitivos válidos son:

Tipo	*Tamaño*
WORD	16 bits
DWORD	32 bits
FLOAT	IEEE float
DOUBLE	64 bits
CHAR	8 bits
UCHAR	8 bits
BYTE	8 bits
STRING	Cadena de caracteres acabada en NULL
CSTRING	Cadena de caracteres con formato C (actualmente sin soporte)
UNICODE	Cadena de caracteres UNICODE(actualmente sin soporte)

Los tipos de datos adicionales definidos por plantillas encontradas anteriormente en la secuencia de datos se pueden también referir dentro de una definición de la plantilla. No se permiten referencias adelantadas.
Cualquier tipo de datos válido puede ser expresado como un array en la definición de plantilla. La sintaxis básica es:

array <tipo_de_datos> <nombre>[<dimensión-tamaño>];

donde <dimensión_tamaño> puede ser un entero o una referencia a otro miembro de plantilla por cuyo valor se sustituye.
Los arrays pueden ser n-dimensionales, donde n se determina por el número de pares de paréntesis cuadrados que aparecen en la secuencia. Por ejemplo:
array DWORD FixedHerd[24];
array DWORD Herd[nCows];
array FLOAT Matrix4x4[4][4];

Restricciones: Las plantillas pueden ser *abiertas, cerradas o restringidas*. Estas restricciones determinan qué tipos de datos pueden aparecer en la jerarquía de un objeto de datos definido por la plantilla. Una plantilla abierta no tiene restricciones, una cerrada rechaza todos los tipos de datos y una restringida permite una lista de tipos de datos declarada. La sintaxis es como sigue:

Tres puntos encerrados por paréntesis cuadrados indican una plantilla abierta [**...**]
Un lista de tipos de datos declarados separados por comas, seguidos opcionalmente por sus uuids encerrados por paréntesis cuadrados indica una plantilla restringida
[{ data-type [UUID] , }…]
La ausencia de las dos formas anteriores indica una plantilla cerrada.
Ejemplo:

```
template Mesh {
<3D82AB44-62DA-11cf-AB39-0020AF71E433>
DWORD nVertices;
array Vector vertices[nVertices];
DWORD nFaces;
array MeshFace faces[nFaces];
 [ ... ]                          // Plantilla abierta.
}
template Vector {
<3D82AB5E-62DA-11cf-AB39-0020AF71E433>
FLOAT x;
FLOAT y;
FLOAT z;
}                                 // Plantilla cerrada.
template FileSystem {
<UUID>
STRING name;
[ Directory <UUID>, File <UUID> ]          // Plantilla restringida.
}
```

Hay una plantilla especial, la plantilla de cabecera. Es recomendable que cada aplicación defina esta plantilla y la use para definir información específica de aplicación como la versión. Si está presente, la cabecera debe ser leída por el API de formato de fichero y si está disponible un miembro *flags*, se utilizará para determinar cómo interpretar la información siguiente. El miembro *flags*, si está definido, debe ser un DWORD. Un bit está actualmente definido, el bit 0, si esté bit vale 0,los datos siguientes en el fichero son binarios; si fuese 1, los datos estarían en formato texto. Se pueden usar varias cabeceras de datos para cambiar entre datos binarios y en formato texto dentro de un mismo archivo.

Datos:
Los objetos de datos contienen los datos reales o una referencia a estos datos. Cada uno tiene su correspondiente plantilla que especifica el tipo de datos.
La forma de estos objetos de datos es la siguiente:

```
<Identificador> [nombre] {
<miembro 1>;
…
<miembro n>;
}
```

Identificador: Es opcional y debe coincidir con un tipo de datos previamente definido o un tipo de datos primitivo.
Nombre: Opcional.

Miembros: Pueden ser los siguientes.

Objeto de datos: Un objeto jerarquizado de los datos. Esto permite que la naturaleza jerárquica del formato del archivo sea expresada. Los tipos de objetos de datos jerarquizados permitidos en la jerarquía pueden ser restringidos.

Referencia a datos: Una referencia a un objeto de datos previamente encontrado. Su sintaxis es: {nombre}

Lista de enteros: Una lista de enteros separados por puntos y comas.

Lista de reales en punto flotante: Una lista de floats separados por puntos y comas.

Lista de strings: Una lista de Strings separados por puntos y comas.

Uso de las comas y los puntos y coma.

Este puede ser el tema más complejo de la sintaxis del formato de archivos, pero es el más estricto. Las comas se usan para separar miembros de un array, los puntos y coma se usan para terminar cada icono de datos.

Por ejemplo, tenemos una plantilla definida como:

```
template foo {
DWORD bar;
}
```

una instancia de esta será de la forma:

```
foo dataFoo {
1;
}
```

Ahora, una plantilla que contiene otra plantilla:

```
template foo {
DWORD bar;
DWORD bar2;
}
template container {
FLOAT aFloat;
foo aFoo;
}
```

Entonces una instancia será:

```
container dataContainer {
1.1;
2; 3;;
}
```

Atención en que la segunda línea que representa a foo dentro de container tiene dos punto y coma al final de la linea. El primero indica el final del icono aFoo, y el segundo indica el final de container.

8.5.1 Plantillas definidas en el API de DIRECT3D

	Nombre de plantilla	***UUID***	
	Header	<3D82AB43-62DA-11cf-AB39-0020AF71E433>	
Nombre de miembro	***Tipo***	***Tamaño de array opcional***	***Tamaño de datos opcional***
major minor flags	WORD WORD DWORD		Ninguno
Descripcion			
Esta plantilla define la cabecera específica de la aplicación DirectX. Este modo utiliza los flags principal(major) y secundario(minor).			

	Nombre de plantilla	***UUID***	
	Vector	<3D82AB5E-62DA-11cf-AB39-0020AF71E433>	
Nombre de miembro	***Tipo***	***Tamaño de array opcional***	***Tamaño de datos opcional***
X Y Z	FLOAT FLOAT FLOAT		Ninguno
Descripción			
Esta plantilla define un vector.			

	Nombre de plantilla	***UUID***	
	Coords2d	<F6F23F44-7686-11cf-8F52-0040333594A3>	
Nombre de miembro	***Tipo***	***Tamaño de array opcional***	***Tamaño de datos opcional***
U V	FLOAT FLOAT		Ninguno
Descripción			
Un vector bidimensional usado para definir las coordenadas de la textura de una malla..			

	Nombre de plantilla	***UUID***	
	Quaternion	<10DD46A3-775B-11cf-8F52-0040333594A3>	
Nombre de miembro	***Tipo***	***Tamaño de array opcional***	***Tamaño de datos opcional***
s v	FLOAT Vector		Ninguno
Descripción			
Actualmente sin uso			

	Nombre de plantilla	***UUID***	
	Matrix4x4	<F6F23F45-7686-11cf-8F52-0040333594A3>	
Nombre de miembro	***Tipo***	***Tamaño de array opcional***	***Tamaño de datos opcional***
matrix	array FLOAT	16	Ninguno
Descripción			
Define una matriz 4 x 4. Se usa como un matriz de transformación.			

	Nombre de plantilla	***UUID***	
	ColorRGBA	<35FF44E0-6C7C-11cf-8F52-0040333594A3>	
Nombre de miembro	***Tipo***	***Tamaño de array opcional***	***Tamaño de datos opcional***
red green blue alpha	FLOAT FLOAT FLOAT FLOAT		Ninguno
Descripción			
Define un objeto color con un componente alfa. Se usa para establecer el color de una cara en la definición de la plantilla material.			

	Nombre de plantilla	***UUID***	
	ColorRGB	<D3E16E81-7835-11cf-8F52-0040333594A3>	
Nombre de miembro	***Tipo***	***Tamaño de array opcional***	***Tamaño de datos opcional***
red green blue	FLOAT FLOAT FLOAT		Ninguno
Descripción			
Define un objeto de color básico RGB			

	Nombre de plantilla	***UUID***
	Indexed Color	<1630B820-7842-11cf-8F52-0040333594A3>
Nombre de miembro	***Tipo***	***Tamaño de array opcional***
index ColorRGBA	DWORD indexColor	
Descripción		
Consiste en un parámetro índice y un color RGBA y se usa para definir colores de vertices de una malla. El índice define el vertice en el que se aplica el color.		

	Nombre de plantilla	***UUID***	
	Boolean	<4885AE61-78E8-11cf-8F52-0040333594A3>	
Nombre de miembro	***Tipo***	***Tamaño opcional de array***	***Tamaño de datos opcional***
WORD	truefalse		Ninguno
Descripción			
Define un tipo booleano.			

	Nombre de plantilla	***UUID***	
	Boolean2d	<4885AE63-78E8-11cf-8F52-0040333594A3>	
Nombre de miembro	***Tipo***	***Tamaño opcional de array***	***Tamaño opcional de datos***
u v	Boolean Boolean		Ninguno
Descripción			
Define un conjunto de dos booleanos usados en la plantilla MeshFaceWraps para definir la topología de la textura de una cara individual.			

	Nombre de plantilla	***UUID***	
	Material	<3D82AB4D-62DA-11cf-AB39-0020AF71E433>	
Nombre de miembro	***Tipo***	***Tamaño opcional de array***	***Objeto de datos opcional***
faceColor power specularColor emissiveColor	ColorRGBA FLOAT ColorRGB ColorRGB		Alguno
Descripción			
Define un color material básico que puede aplicarse a una malla completa o a caras individuales de una malla. El miembro power especifica la componente especular del material. La luz ambiente requiere un componente alfa.			
Objetos de datos opcionales usados por Direct3DRM			
TextureFilename	Si no aparece, la malla está sin texturar		

	Nombre de plantilla	***UUID***	
	TextureFilename	<A42790E1-7810-11cf-8F52-0040333594A3>	
Nombre de miembro	***Tipo***	***Tamaño opcional de array***	***Objeto de datos opcional***
filename	STRING		Ninguno
Descripción			
Permite especificar el nombre de archivo de una textura a aplicar a una malla o una cara. Debería aparecer dentro de un objeto material.			

	Nombre de plantilla	***UUID***	
	MeshFace	<3D82AB5F-62DA-11cf-AB39-0020AF71E433>	
Nombre de miembro	***Tipo***	***Tamaño opcional de array***	***Objeto de datos opcional***
nFaceVertexIndices faceVertexIndices	DWORD array DWORD	 NFaceVertexIndices	Ninguno
Descripción			
Se usa por la plantilla Mesh para definir las caras de las mallas. Cada elemento del array nFaceVertexIndices referencia un vertice de la malla usado para construir la cara.			

	Nombre de plantilla	***UUID***	
	MeshFaceWraps	<4885AE62-78E8-11cf-8F52-0040333594A3>	
Nombre de miembro	***Tipo***	***Tamaño opcional de array***	***Objeto de datos opcional***
nFaceWrapValues FaceWrapValues	DWORD Boolean2d		Ninguno
Descripción			
Esta plantilla se usa para definir la topología de la textura de cada cara . nFaceWrapValues debe ser igual al numero de caras en la malla.			

	Nombre de plantilla	***UUID***	
	MeshTextureCoords	<F6F23F40-7686-11cf-8F52-0040333594A3>	
Nombre de miembro	***Tipo***	***Tamaño opcional de array***	***Objeto de datos opcional***
nTextureCoords textureCoords	DWORD array Coords2d	 nTextureCoords	Ninguno
Descripción			
Define las coordenadas de textura de una malla.			

	Nombre de plantilla	***UUID***	
	MeshNormals	<F6F23F43-7686-11cf-8F52-0040333594A3>	
Nombre de miembro	***Tipo***	***Tamaño opcional de array***	***Objeto de datos opcional***
nNormals normals nFaceNormals faceNormals	DWORD array Vector DWORD array MeshFace	 nNormals nFaceNormals	Ninguno
Descripción			
Define normales para una malla.El primer array de vectores son los vectores de normales, y el segundo es un array de índices especificando que normales deben aplicarse a la cara dada. nFaceNormals debe ser igual al numero de caras en una malla.			

	Nombre de plantilla	***UUID***	
	MeshVertexColors	<1630B821-7842-11cf-8F52-0040333594A3>	
Nombre de miembro	***Tipo***	***Tamaño opcional de array***	***Objeto de datos opcional***
nVertexColors vertexColors	DWORD array IndexedColor	 nVertexColors	Ninguno
Descripción			
Especifica el color de los vértices en una malla, en oposición a aplicar un material por cara o por malla.			

	Nombre de plantilla	***UUID***	
	MeshMaterialList	<F6F23F42-7686-11cf-8F52-0040333594A3>	
Nombre de miembro	***Tipo***	***Tamaño opcional de array***	***Objeto de datos opcional***
nMaterials nFaceIndexes FaceIndexes	DWORD DWORD array DWORD	 nFaceIndexes	Material
Descripción			
Se usa en un objeto Mesh para especificar qué material aplicar a qué caras. nMaterials especifica cuántos materales están presentes, y el objeto Material especifica qué material aplicar.			

	Nombre de plantilla	***UUID***	
	Mesh	<3D82AB44-62DA-11cf-AB39-0020AF71E433>	
Nombre de miembro	***Tipo***	***Tamaño opcional de array***	***Objeto de datos opcional***
nVertices vertices nFaces faces	DWORD array Vector DWORD array MeshFace	 nVertices nFaces	Alguno
Descripción			
Define una malla simple.El primer array es una lista de vértices y el segundo define las caras de la malla indexadas por el array de vértices.			
Objetos de datos opcionales usados por Direct3DRM			
MeshFaceWraps	Si no está presente, u y v se ponen por defecto a falso.		
MeshTextureCoords	Si no aparece, no hay coordenadas de textura.		
MeshNormals	Si no aparece las normales se generan usando el método GenerateNormals() del API.		
MeshVertexColors	Si no aparece, los colores se ponen por defecto a blanco.		
MeshMaterialList	Si no aparece, el material se pone por defecto a blanco.		

	Nombre de plantilla	***UUID***	
	FrameTransformMatrix	<F6F23F41-7686-11cf-8F52-0040333594A3>	
Nombre de miembro	***Tipo***	***Tamaño opcional de array***	***Objeto de datos opcional***
frameMatrix	Matrix4x4		Ninguno
Descripción			
Define una transformación local para un frame(y sus hijos).			

	Nombre de plantilla	***UUID***	
	Frame	<3D82AB46-62DA-11cf-AB39-0020AF71E433>	
Nombre de miembro	***Tipo***	***Tamaño opcional de array***	***Objeto de datos opcional***
Ninguno			Alguno
Descripción			
Define un frame. Puede contener objetos de tipo Mesh y un FrameTransformMatrix.			
Objetos de datos opcionales usados por Direct3DRM			
FrameTransformMatrix	Si no aparece, no se aplicarán transformaciones locales al frame.		
Mesh	Cualquier numero de objetos Mesh se convertirán en hijos del frame.		

	Nombre de plantilla	***UUID***	
	FloatKeys	<10DD46A9-775B-11cf-8F52-0040333594A3>	
Nombre de miembro	***Tipo***	***Tamaño opcional de array***	***Objeto de datos opcional***
nValues values	DWORD array FLOAT	 nValues	Ninguno
Descripción			
Define un array de floats y el número de floats en el array. Se usa para definir conjuntos de claves de animación.			

	Nombre de plantilla	***UUID***	
	TimedFloatKeys	<F406B180-7B3B-11cf-8F52-0040333594A3>	
Nombre de miembro	***Tipo***	***Tamaño opcional de array***	***Objeto de datos opcional***
time tfkeys	DWORD FloatKeys		Ninguno
Descripción			
Define un conjunto de floats y un tiempo positivo usado en animaciones.			

	Nombre de plantilla	***UUID***	
	AnimationKey	<10DD46A8-775B-11cf-8F52-0040333594A3>	
Nombre de miembro	***Tipo***	***Tamaño opcional de array***	***Objeto de datos opcional***
keyType nKeys keys	DWORD DWORD array TimedFloatKeys	 NKeys	Ninguno
Descripción			
Define un conjunto de claves de animaciones.s.El parámetro keyType especifica si las claves son de rotación, escalación o posición (usando los enteros 0, 1 o 2 respectivamente).			

	Nombre de plantilla	***UUID***	
	AnimationOptions	<E2BF56C0-840F-11cf-8F52-0040333594A3>	
Nombre de miembro	***Tipo***	***Tamaño opcional de array***	***Objeto de datos opcional***
Openclosed Positionquality	DWORD DWORD		Ninguno
Descripción			
Permite establecer las opciones de animacion de D3DRM.El parámetro openclosed puede ser 0 para cerrado o 1 para una animación abierta. El parámetro positionquality se usa para establecer la calidad de cualquier clave de posición especificada y puede ser 0 para para posiciones spline o 1 para posiciones lineales. Por defecto una animación es abierta y usa claves de posición lineales.			

	Nombre de plantilla	***UUID***	
	Animation	<3D82AB4F-62DA-11cf-AB39-0020AF71E433>	
Nombre de miembro	***Tipo***	***Tamaño opcional de array***	***Objeto de datos opcional***
Ninguno			Alguno
Descripción			
Contiene animaciones que referencian un frame previo. Debe contener una referencia a un frame y por lo menos un conjunto de AnimationKeys. Puede tener también un objeto AnimationOptions.			
Objetos de datos opcionales usados por Direct3DRM			
AnimationKey	Una animación no tiene significado sin AnimationKeys.		
AnimationOptions	Si no aparece, la animación es abierta y usa claves de posición lineales.		

	Nombre de plantilla	***UUID***	
	AnimationSet	<3D82AB50-62DA-11cf-AB39-0020AF71E433>	
Nombre de miembro	***Tipo***	***Tamaño opcional de array***	***Objeto de datos opcional***
Ninguno			Animation
Descripción			
Conriene uno o más objetos Animation y es equivalente al concepto de conjuntos de animación de D3DRM. Lo que significa que cada animación dentro de un conjunto de animaciones tiene el mismo tiempo en cualquier punto dado. Incrementando el tiempo del conjunto de animación se incrementará el tiempo de todas las animaciones que contiene.			

9. FMOD

Aquí se explica cómo utilizar las librerías de Fmod para reproducir sonidos desde una aplicación.

9.1 ¿QUÉ ES FMOD?

FMOD es una librería multiplataforma para el manejo de sonido y música en videojuegos, demos, aplicaciones multimedia, etc. Está orientada especialmente para su uso en la programación de videojuegos. Algunas de las plataformas donde se encuentra disponible son: Windows, Linux, Windows CE, y Macintosh, incluso está disponible para algunas consolas como por ejemplo: GameCube, PS2 y XBOX. Esta librería es gratuita siempre y cuando nuestras aplicaciones no sean comerciales, en caso contrario se debe pagar una licencia. Las últimas versiones de esta librería se pueden encontrar en su sitio oficial http://www.fmod.org

Las versiones para consolas no pueden ser descargadas directamente desde el sitio, ya que para utilizarlas se debe acreditar ser un desarrollador de Microsoft, Nintendo o Sony.

Otra de las ventajas de esta librería es que está soportada por varios compiladores, entre ellos tenemos a Microsoft Visual C++ 5 & 6, Watcom, Borland, LCC-WIN32, Metrowerks/CodeWarrior, GCC, MinGW & CygWin, Delphi, Kylix, Visual Basic.
Algunos de los formatos soportados por esta librería son los siguientes: MOD, S3M, XM, IT y MID para música; WAV, MP2, MP3 Y OGG para sonidos.

9.2 INICIALIZACIÓN DE LA LIBRERÍA

Lo primero que debemos colocar en nuestro programa es un include de la cabecera fmod.h

#include <fmod.h>

Para poder usar fmod debemos inicializar el sistema de sonido, esto lo hace la función **FSOUND_Init**, veamos su prototipo:

signed char F_API FSOUND_Init(int mixrate, int maxsoftwarechannels, unsigned int flags);

El parámetro *mixrate* corresponde a la frecuencia, la cual puede variar entre los 4000 y 65535 HZ. Lo común será pasarle 44100, con lo que obtendremos una muy buena calidad de sonido. Luego tenemos el parámetro *maxsoftwarechannels* que corresponde al número máximo de canales que pueden existir, un buen número que se le puede pasar es 32. Por último tenemos el parámetro *flags*, el cual activa otras características para el sonido. La función devuelve true si todo salió bien y false si hubo algún error.

Una cosa importante es que antes de salir de nuestro programa tendremos que cerrar el sistema de sonido, esto lo haremos con la función **FSOUND_Close**:

void F_API FSOUND_Close();

9.3 REPRODUCIENDO SONIDOS

Lo primero que tenemos que saber es en dónde almacenar los datos del sonido, para esto existe una estructura llamada **FSOUND_SAMPLE**. La librería FMOD dispone de varias funciones para manejar el sonido, nosotros usaremos la siguiente para cargar los sonidos en esta estructura:

FSOUND_SAMPLE * F_API FSOUND_Sample_Load(int index, const char *name, unsigned int inputmode, int memlength);

En el primer parámetro *index*, debemos pasarle el número del canal donde cargaremos el sonido, lo mejor es pasarle **FSOUND_FREE**, entonces el programa elegirá un canal libre arbitrariamente. Luego tenemos el parámetro *name* que corresponde al nombre del archivo que queremos cargar. En el tercer parámetro *inputmode*, colocaremos el formato de los datos, existe una gran lista de estos modos, nosotros utilizaremos el siguiente: **FSOUND_STEREO**. Por último tenemos a *memlength*, aquí pasaremos el valor 0 ya que el sonido lo estamos cargando desde un archivo, si fuera desde la memoria tendríamos que indicarle la longitud en bytes del sonido. La función retorna un puntero a FSOUND_SAMPLE o NULL si hubo algún error.

Para liberar la memoria ocupada por este sonido y a la vez dejar nuevamente este canal disponible, usaremos la siguiente función:

void F_API FSOUND_Sample_Free(FSOUND_SAMPLE *_sptr_);

Simplemente se le pasa como parámetro un puntero a la estructura que se quiere eliminar de la memoria.

Una vez que tenemos nuestro sonido en memoria, podremos reproducirlo. Esto lo haremos a través de la función:

int F_API FSOUND_PlaySound(int channel, FSOUND_SAMPLE *sptr);

En el parámetro *channel*, indicaremos el número del canal en el cual reproduciremos el sonido, nosotros le pasaremos **FSOUND_FREE**.
Por último tenemos el puntero *sptr*, que corresponde al sonido que reproduciremos. La función devuelve el número del canal seleccionado, es conveniente guardarlo en alguna variable ya que lo utilizaremos más adelante con otras funciones.

Para detener el sonido que se está reproduciendo usaremos la función:

signed char F_API FSOUND_StopSound(int channel);

Simplemente le pasaremos el número del canal que queremos dejar de reproducir. La función devuelve true si todo resultó correctamente o false en caso contrario.

También existe una función para colocar en pausa el sonido que se está reproduciendo:

signed char F_API FSOUND_SetPaused(int channel, signed char paused);

Aquí le pasamos el número del canal, y en el segundo parámetro true si queremos colocarlo en pausa, o false si deseamos que continúe reproduciéndose. Devuelve true si todo salió bien o false en caso contrario.

Si queremos modificar el volumen del sonido usaremos la siguiente función:

signed char F_API FSOUND_SetVolume(int channel, int vol);

Le pasamos el número del canal, y en el segundo parámetro el valor del nuevo volumen que puede variar entre 0 y 255. Devuelve true si todo salió bien o false en caso contrario.

La función inversa a la anterior es:

int F_API FSOUND_GetVolume(int channel);

La cual nos devuelve el volumen actual del canal que debe ser pasado por parámetro.

También podemos modificar el paneo en un canal determinado mediante la función:

signed char F_API FSOUND_SetPan(int channel, int pan);

Se le debe pasar el número del canal, y el nuevo valor de paneo que debe estar entre 0 (completamente a la izquierda) o 255 (completamente a la derecha).

Si deseamos saber el valor del paneo actual usaremos la siguiente función:

int F_API FSOUND_GetPan(int channel);

Se le debe pasar el número del canal y retornará el valor del paneo actual.
Siguiendo con las funciones, también disponemos de una que nos permite silenciar un canal determinado, esto lo haremos con la siguiente función:

signed char F_API FSOUND_SetMute(int channel, signed char mute);
El primer parámetro es el número del canal, y el segundo true si deseamos activarlo o false si lo queremos desactivar. Devuelve true si todo resultó bien o false en caso contrario.

Por ultimo veremos una función que nos permite activar o desactivar un loop de un sonido:

signed char F_API FSOUND_SetLoopMode(int channel, unsigned int loopmode);
Se le debe pasar el número del canal, y en el segundo parámetro FSOUND_LOOP_NORMAL si queremos activarlo o FSOUND_LOOP_OFF si lo vamos a desactivar. Devuelve true si toda ha salido bien o false en caso contrario.

9.4 REPRODUCIENDO MÚSICA

Al igual que sucede con los sonidos, la librería FMOD tiene una estructura para almacenar la música, esta es **FMUSIC_MODULE**. Existen funciones muy similares a las hemos visto en el apartado de sonidos para manejar la música, el único detalle es aquí no existen los canales, así que no nos preocuparemos por eso.

Ahora veremos un listado de los prototipos de las funciones más utilizadas. Su utilidad es similar a las anteriores funciones, lo único que no hacen uso del concepto de canal. Las funciones son las siguientes:

FMUSIC_MODULE * F_API FMUSIC_LoadSong(const char *name);
signed char F_API FMUSIC_FreeSong(FMUSIC_MODULE *mod);
signed char F_API FMUSIC_PlaySong(FMUSIC_MODULE *mod);
signed char F_API FMUSIC_SetPaused(FMUSIC_MODULE *mod, signed char pause);
signed char F_API FMUSIC_StopSong(FMUSIC_MODULE *mod);
signed char F_API FMUSIC_SetMasterVolume(FMUSIC_MODULE *mod, int volume);
int F_API FMUSIC_GetMasterVolume(FMUSIC_MODULE *mod);
signed char F_API FMUSIC_SetLooping(FMUSIC_MODULE *mod, signed char looping);

Opciones de preinicialización

Antes de iniciar fmod para poder reproducir los sonidos que queremos, debemos tener en cuenta varios aspectos.

Antes de llamar a FSOUND_Init() podemos establecer el tipo de salida que queremos. Esto lo podemos hacer con la llamada a:

signed char F_API FSOUND_SetOutput(int outputtype);
Donde el parámetro outputtype puede ser uno de los componentes del siguiente tipo:

```
enum FSOUND_OUTPUTTYPES
{
    FSOUND_OUTPUT_NOSOUND,    /* Sin sonido, todas las llamadas an éxito pero no hacen nada*/
    FSOUND_OUTPUT_WINMM,      /* Windows Multimedia driver. */
    FSOUND_OUTPUT_DSOUND,     /* DirectSound driver.  Es necesario para soportar EAX2, EAX3 o FX. */
    FSOUND_OUTPUT_A3D,        /* A3D driver. */
    FSOUND_OUTPUT_OSS,        /* Linux/Unix OSS (Open Sound System) driver, es decir, los drivers del kernel.*/
```

```
        FSOUND_OUTPUT_ESD,      /* Linux/Unix ESD (Enlightment Sound
Daemon) driver. */
        FSOUND_OUTPUT_ALSA,     /* Linux Alsa driver. */
        FSOUND_OUTPUT_ASIO,     /* Low latency ASIO driver */
        FSOUND_OUTPUT_XBOX,     /* Xbox driver */
        FSOUND_OUTPUT_PS2,      /* PlayStation 2 driver */
        FSOUND_OUTPUT_MAC,      /* Mac SoundManager driver */
        FSOUND_OUTPUT_GC,       /* Gamecube driver */
        FSOUND_OUTPUT_NOSOUND_NONREALTIME   /* Es igual que
nosound, pero la generación de sonido la lleva a cabo FSOUND_Update */
    };
```

9.5 ARCHIVOS MP3

Para reproducir archivos mp3 hay que tratarlos de diferente manera a lo hecho hasta ahora. Necesitamos un puntero al tipo FSOUND_STREAM. Para abrir el archivo mp3 tenemos que llamar al método:

FSOUND_STREAM* F_API FSOUND_Stream_Open(const char *name_or_data, unsigned int mode, int offset, int length);

Donde el parámetro name_or_data indica el nombre del archive o la posición de memoria de la cual leer los datos. El parámetro mode indica el modo de abrir el archivo, si queremos leer desde memoria habrá que indicarlo con FSOUND_LOADMEMORY. Una vez que tenemos el archivo cargado en el puntero a FSOUND_STREAM podemos empezar la reproducción del mismo llamando a:

Int F_API FSOUND_Stream_Play(int channel, FSOUND_STREAM *stream);

Si le indicamos en el parámetro channel FSOUND_FREE el programa elegirá un canal libre arbitrariamente.

Para parar la reproducción y liberar el canal llamamos a la función:

signed char F_API FSOUND_Stream_Close(FSOUND_STREAM *stream);

9.6 IMPLEMENTACIÓN

En nuestro programa la clase que se encarga de lo concerniente a los sonidos es la clase GestorSonido. Esta clase inicializa la librería de fmod para poder reproducir sonidos. Puede reproducir un archivo mp3, sólo uno a la vez. Un archivo de música por medio de FMUSIC_PlaySong y una serie de FSOUND_SAMPLE. Esta clase consta de dos listas, una con los nombres de los archivos de los FSOUND_SAMPLE a usar y otra con los FSOUND_SAMPLE cargados. Cuando una clase quiere cargar un FSOUND_SAMPLE para poder utilizarlo más adelante le pide al gestor que lo cargue dándole el nombre del archivo. El gestor busca en la lista de nombres si ya estaba el FSOUND_SAMPLE, si estaba ya, le devuelve la posición del FSOUND_SAMPLE, y si no estaba lo añade al final y le da su posición. De esta forma evitamos tener varios FSOUND_SAMPLE iguales cargados en memoria. Cuando una clase quiere reproducir un FSOUND_SAMPLE lo único que hace es llamar al gestor y decirle la posición del FSOUND_SAMPLE que quiere reproducir, el gestor lo saca de la lista y lo reproduce.

Para reproducir música o un archivo mp3, la clase que quiere ejecutarlo llama al gestor para que reproduzca el archivo cuyo nombre le pasa como parámetro.

Una cosa a tener en cuenta es que GestorSonido no es una clase, es un namespace, de esta forma solo hay una copia del gestor en ejecución.

Al realizar la búsqueda en la lista de nombres se pensó en hacer búsqueda dicotómica y una inserción ordenada. Pero surgió el problema de que devolvíamos la posición del FSOUND_SAMPLE, si hacíamos una inserción ordenada dicha posición podría variar según insertamos más elementos. Las dos soluciones que encontramos fueron hacer búsqueda lineal al insertar nuevos elementos y hacer un acceso constante a la hora de reproducir los FSOUND_SAMPLES , o bien hacer una búsqueda dicotómica al insertar elementos y volver a hacer búsqueda dicotómica al reproducir los sonidos. Elegimos el primer camino ya que la búsqueda lineal se hace sólo al cargar los FSOUND_SAMPLES, y los accesos constantes ocurren durante toda la ejecución

10. SOCKETS

Esta sección trata sobre cómo controlar el objeto a tratar en el simulador desde otros programa mediante el uso de sockets. Se explica cómo implementar y usar sockets así como la integración de clases c++ en Matlab (Simulink).

10.1 SOCKETS

Los sockets son puntos finales de enlaces de comunicaciones entre procesos. Los procesos los tratan como descriptores de ficheros, de forma que se pueden intercambiar datos con otros procesos transmitiendo y recibiendo a través de sockets. El tipo de sockets describe la forma en la que se transfiere información a través de ese socket.

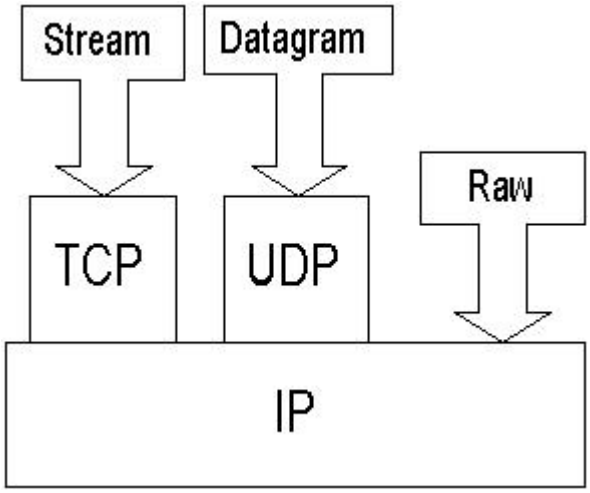


Figura 10.1 Esuqema de las capas de los sockets

10.1.1 Sockets Stream (TCP, Transport Control Protocol)

Son un servicio orientado a conexión donde los datos se transfieren sin encuadrarlos en registros o bloques. Si se rompe la conexión entre los procesos, éstos serán informados.
El protocolo de comunicaciones con streams es un protocolo orientado a conexión, ya que para establecer una comunicación utilizando el protocolo TCP, hay que establecer en primer lugar una conexión entre un par de sockets. Mientras uno de los sockets atiende peticiones de conexión (servidor), el otro solicita una conexión (cliente). Una vez que los dos sockets estén conectados, se pueden utilizar para transmitir datos en ambas direcciones.

10.1.2 Sockets Datagrama (UDP, User Datagram Protocol)

Son un servicio de transporte sin conexión. Son más eficientes que TCP, pero no está garantizada la fiabilidad. Los datos se envían y reciben en paquetes, cuya entrega no está garantizada. Los paquetes pueden ser duplicados, perdidos o llegar en un orden diferente al que se envió.
El protocolo de comunicaciones con datagramas es un protocolo sin conexión, es decir, cada vez que se envíen datagramas es necesario enviar el descriptor del socket local y la dirección del socket que debe recibir el datagrama. Como se puede ver, hay que enviar datos adicionales cada vez que se realice una comunicación.

10.1.3 Sockets Raw

Son sockets que dan acceso directo a la capa de software de red subyacente o a protocolos de más bajo nivel. Se utilizan sobre todo para la depuración del código de los protocolos.

10.1.4 Diferencias entre Sockets Stream y Datagrama

Ahora se nos presenta un problema, ¿qué protocolo, o tipo de sockets, debemos usar - UDP o TCP? La decisión depende de la aplicación cliente/servidor que estemos escribiendo. Vamos a ver algunas diferencias entre los protocolos para ayudar en la decisión.

En UDP, cada vez que se envía un datagrama, hay que enviar también el descriptor del socket local y la dirección del socket que va a recibir el datagrama, luego éstos son más grandes que los TCP. Como el protocolo TCP está orientado a conexión, tenemos que establecer esta conexión entre los dos sockets antes de nada, lo que implica un cierto tiempo empleado en el establecimiento de la conexión, que no existe en UDP.

En UDP hay un límite de tamaño de los datagramas, establecido en 64 kilobytes, que se pueden enviar a una localización determinada, mientras que TCP no tiene límite; una vez que se ha establecido la conexión, el par de sockets funciona como los streams: todos los datos se leen inmediatamente, en el mismo orden en que se van recibiendo.

UDP es un protocolo desordenado, no garantiza que los datagramas que se hayan enviado sean recibidos en el mismo orden por el socket de recepción. Al contrario, TCP es un protocolo ordenado, garantiza que todos los paquetes que se envíen serán recibidos en el socket destino en el mismo orden en que se han enviado.

Los datagramas son bloques de información del tipo lanzar y olvidar. Para la mayoría de los programas que utilicen la red, el usar un flujo TCP en vez de un datagrama UDP es más sencillo y hay menos posibilidades de tener problemas. Sin embargo, cuando se requiere un rendimiento óptimo, y está justificado el tiempo adicional que supone realizar la verificación de los datos, los datagramas son un mecanismo realmente útil.

En resumen, TCP parece más indicado para la implementación de servicios de red como un control remoto (rlogin, telnet) y transmisión de ficheros (ftp); que necesitan transmitir datos de longitud indefinida. UDP es menos complejo y tiene una menor sobrecarga sobre la conexión; esto hace que sea el indicado en la implementación de aplicaciones cliente/servidor en sistemas distribuidos montados sobre redes de área local.

Debido a las características de los dos tipos de sockets optamos por usar sockets TCP, ya que nos asegura que los datos van a llegar en orden y pueden tener longitud variable. En conclusión, como ya se ha comentado antes, TCP es más indicado para un

control remoto. De todas formas, el archivo "PracticalSocket" tiene implementados los dos tipos de sockets TCP y UDP.

10.2 USO DE SOCKETS

En nuestro programa hemos tenido en cuenta la demanda del usuario para usar o no sockets. Se lee mediante LUA el archivo "Control.lua". La variable socket indicará si se ha de usar sockets o no. A parte de esto, también se leen la dirección y el puerto donde queremos que se cree el socket.

10.2.1 Creación y uso de sockets

Para crear un socket servidor lo hacemos mediante el constructor siguiente:

TCPServerSocket(const string &localAddress, unsigned short localPort, int queueLen = 5) throw(SocketException);

La dirección se indica en localAddress, el puerto en localPort y el parámetro queueLen indica el número máximo de peticiones que tratará el servidor. Por defecto se ha establecido en 5, aunque realmente sólo habrá un único controlador.

Una vez que tenemos los datos creamos un socket de tipo servidor en la dirección y puerto indicados y lo ponemos a la escucha para recibir peticiones de conexión. Esto se hace mediante el método accept() que bloquea la ejecución hasta que recibe una petición o se produce un error. Cuando recibe una petición crea un nuevo socket que será el camino por donde tendremos que enviar y recibir los datos. Esta función devuelve el nuevo socket. De esta manera hay que ejecutar primero el simulador y luego el programa que actúa como controlador. Debido a que accept() bloque la ejecución, si se pierde la conexión por alguna razón habrá que iniciar el programa de nuevo, no llamamos a accept() de nuevo porque el programa se quedaría bloqueado.

TCPSocket *accept() throw(SocketException);

En este punto le toca al controlador crear un socket cliente con la misma dirección y puerto que el servidor. Al crear dicho socket, si todo se hace sin errores, el servidor aceptará la conexión pedida por el socket cliente

Ya estamos listos para enviar o recibir información a través del socket que acabamos de crear. Hemos optado por el controlador para que sea el primero en enviar datos, de esta forma podemos establecer unos valores iniciales del objeto a manejar. Los datos que se envían y se reciben van a tener longitudes distintas. Si vemos la conexión desde el simulador se enviarán paquetes de 16 float que contendrán: las fuerzas, los momentos, la posición del cíclico, la orientación del objeto, las potencias de los rotores y la posición actual del objeto. Se recibirán paquetes de 4 float con la información sobre la nueva posición del cíclico a establecer y las nuevas potencias de los rotores.

Para recibir información usamos el método:

int recv(void *buffer, int bufferLen) throw(SocketException);

Este método lee datos del socket actual, hasta bufferLen bytes, y los almacena en el parámetro buffer. Devuelve el número de bytes leídos, 0 para EOF y -1 si ha habido algún error.

Como hemos comentado antes el primero en recibir los datos es el simulador, después de recibir los datos actualiza las variables del objeto según se le indique y posteriormente actualiza el objeto según el tiempo transcurrido y según los datos del objeto en ese momento.

Una vez que el objeto está actualizado el simulador construye un paquete con los nuevos datos del objeto para enviárselo al controlador. Para enviar información se usa el método:

void send(const void *buffer, int bufferLen) throw(SocketException);

Este método escribe bufferLen bytes del parámetro buffer en el socket actual. La función devuelve trae si se envía con éxito.

10.2.1 Cierre de la conexión

El cierre de la conexión lo hemos implementado en el destructor del socket. Al cerrar el socket el cliente recibirá un 0 que le indica EOF y sabe que la conexión se ha cerrado.

10.3 WINSOCK

Para implementar los sockets nos hemos apoyado en Winsock. Ahora explicaremos los pasos a seguir con Winsock para preparar los sockets.

10.3.1 Inicializar Winsock

Todas las aplicaciones con Winsocl deben ser inicializadas antes de usarse para asegurarnos de que el sistema soporta los sockets de Windows.

Primero hay que crear un objeto de tipo WSADATA y otro de tipo WORD:

```
WSADATA wsaData;
WORD wVersionRequested = MAKEWORD(2, 0);
```

MAKEWORD(2,0) indica un requisito de versión, en este caso se pide Winsock v2.0
Luego llamamos a WSAStartup con dicho objeto y comprobamos el entero que nos devuelve para chequear errores:

```
if (WSAStartup(wVersionRequested, &wsaData) != 0) {//Cargar WinSock DLL
  throw SocketException("No se ha podido cargar WinSock DLL");
}
```

10.3.2 Creación de un socket (Servidor o Cliente)

Creamos un descriptor del socket, su tipo y el protocolo que va a utilizar:

```
int sockDesc;
int type;
int protocol;
```

Creamos un nuevo socket:

```
if ((sockDesc = socket(PF_INET, type, protocol)) < 0) {
  throw SocketException("Error al crear Socket (socket())", true);
}
```

En nuestra aplicación el tipo de socket es SOCK_STREAM y el protocolo a utilizar es IPPROTO_TCP. PF_INET le indica que la familia de direcciones hace referencia a Internet.

10.3.3 Establecer socket del servidor

Para que un servidor pueda aceptar conexiones de clientes debe estar ligado a una dirección del sistema. Los siguientes pasos explican como ligar un socket ya creado a una dirección y puerto determinados. Los clientes usarán dicha dirección y puerto para

conectarse al servidor. En el ejemplo suponemos que la dirección se guarda en la variable address y el puerto en port.

```
String address;
Unsigned short port;
```

La estructura sockaddr contiene información referente a la familia de direcciones, la dirección IP y el puerto. La estructura sockaddr_in es un subconjunto de sockaddr y se usa para aplicaciones que usan IP versión 4.

Creamos la estructura y la rellenamos. Primero rellenamos el tipo de familia de direcciones:

```
sockaddr_in addr;
addr.sin_family = AF_INET; //dirección de internet
```

Para obtener la dirección primero resolvemos el nombre contenido en la variable address:

```
hostent *host;  // Resolvemos el nombre
if ((host = gethostbyname(address.c_str())) == NULL) {
  throw SocketException("Error en el nombre (gethostbyname())");
}
```

La estructura hostent es la siguiente:

```
typedef struct hostent {
  char FAR* h_name;
  char FAR  FAR** h_aliases;
  short h_addrtype;
  short h_length;
  char FAR  FAR** h_addr_list;
} hostent;
```

La dirección del host está en h_addr_list[0]. Asignamos la dirección y el puerto:

```
//Asignamos la dirección obtenida a partir del nombre
addr.sin_addr.s_addr = *((unsigned long *) host->h_addr_list[0]);
addr.sin_port = htons(port);     //Asignar el puerto
```

Ligamos el servidor, pasando el socket creado y la estructura sockaddr_in:

```
if (bind(sockDesc, (sockaddr *) &addr, sizeof(sockaddr_in)) < 0) {
  throw SocketException("Error en la dirección y puerto (bind())", true);
}
```

El hecho de que la dirección pueda ser expresada mediante un nombre nos da ciertas ventajas. Por ejemplo, si queremos crear un socket en nuestro sistema podremos poner en vez de la dirección IP completa (por ejemplo 127.85.89.133) podemos poner simplemente localhost.

10.3.4 Escucha en un socket (Servidor)

Después de ligar un socket a una dirección y un puerto del sistema, el servidor debe escuchar en dicha dirección y puerto para tratar las solicitudes de conexión.

```
if (listen(sockDesc, queueLen) < 0) {
  throw SocketException("Error en la escucha (listen())", true);
}
```

El parámetro queueLen le indica el número máximo de conexiones que puede aceptar el servidor.

10.3.5 Aceptar una conexión (Servidor)

Una vez que el servidor está a la escucha debe tratar las solicitudes de conexión.

```
int newConnSD;
if ((newConnSD = accept(sockDesc, NULL, 0)) < 0) {
  throw SocketException("Error al aceptar (accept())", true);
}
```

En el parámetro newConnSD tendremos el descriptor del socket aceptado.

10.3.6 Conectarse a un socket (Cliente)

Para que el cliente se pueda comunicar debe conectarse a un servidor. Para ello debe construir una estructura de tipo sockaddr_in con la información del socket donde está escuchando el servidor (dirección y puerto):

```
if (connect(sockDesc, (sockaddr *) &destAddr, sizeof(destAddr)) < 0) {
  throw SocketException("Error al conectar (connect())", true);
}
```

La variable destAddr tendrá la información del destino donde el servidor está escuchando.

10.3.7 Enviar y recibir datos (Cliente o Servidor)

Para enviar información necesitamos el descriptor del socket por dónde vamos a enviar los datos (sockDesc), los datos (buffer) y el tamaño de los datos (bufferLen).

```
if (send(sockDesc, (raw_type *) buffer, bufferLen, 0) < 0) {
  throw SocketException("Error al enviar (send())", true);
}
```

Para recibir datos se hace de forma análoga a lo anterior:

```
if (recv(sockDesc, (raw_type *) buffer, bufferLen, 0) < 0) {
  throw SocketException("Error al recibir (recv())", true);
}
```

10.4 PRACTICALSOCKET

Este archivo contiene la jerarquía de clases indicada en la figura 10.2.

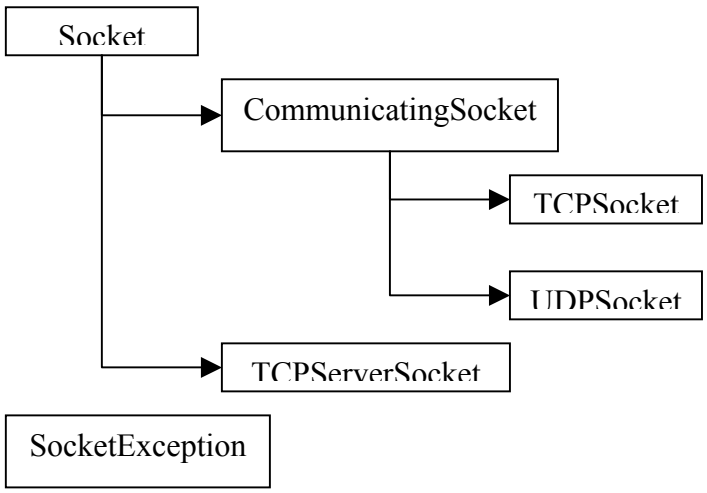


Figura 10.2 Jerarquía de clases deñ archivo “PracticalSocket.h”

Socket es la superclase y contiene el descriptor del socket creado a partir de una dirección y un puerto.
CommunicatingSocket implementa un socket capaz de conectarse, enviar y recibir información.
TCPSocket socket TCP para comunicarse con otros sockets TCP.
TCPServerSocket socket TCP servidor.
UDPSocket socket UDP.

Estas clases hacen más fácil el manejo de sockets por medio de Winsock. Abstraen diversas funciones necesarias para el uso de sockets y permiten al programador realizar conexiones de forma más fácil y práctica.

10.5 CONTROLADOR

Para nuestra aplicación hemos creado otro programa que funciona como un piloto automático para un helicóptero. Es bastante sencillo ya que lo único que hace es mantener el helicóptero en un rumbo hacia el norte.

10.5.1 Controlador desde otro programa de C++

Lo primero que hace el controlador es leer desde un archivo la dirección y el puerto dónde tiene que conectarse mediante un socket. Una vez que se ha conectado envía los datos iniciales del helicóptero. A partir de ahí entra en un bucle donde recibe datos del helicóptero y envía los nuevos datos de control. Este bucle se realiza hasta que el servidor cierra la conexión o se produce algún error en la transmisión de los datos.

Básicamente lo que hace es mantener la guiñada, el cabeceo y el balanceo en ángulos en torno a 0º. Y también mantiene el helicóptero a una altura indicada también el archivo de texto. Hay que tener en cuenta que al ser un sistema de control puede tener cierta oscilación. Hay que llegar a una solución de compromiso entre el tiempo que tarda en llevar el valor de entrada al valor de referencia y la oscilación que queremos que tenga el sistema. En nuestro controlador los valores de entrada se modifican en torno a 0.03 unidades en caso del cíclico, 0.1 unidades en caso del rotor trasero y en 2 unidades en el rotor principal. El valor que sufre mayor oscilación es la altura del helicóptero, pero llega a su valor de referencia en un tiempo menor.

10.5.2 Controlador hecho en Simulink

Se pensó en hacer el control del helicóptero no desde un programa en c++ sino en un programa en Matlab. Matlab está muy orientado a hacer sistemas de control. Nos surgieron muchos problemas a la hora de incluir las librerías de lo sockets en programas c++ compilables por Matlab, pero al final hicimos un controlador en Matlab. El controlador hecho en Matlab hace lo mismo que el programa anterior, pero se puede perfeccionar fácilmente. Para utilizar los sockets en Matlab tuvimos que integrar clases de c++. El sistema que utilizamos para integrar dentro de Matlab clases de c++ fue utilizar los bloques S-Function de la librería “Simulink->User-Defined Functions”. Para utilizar las clases de c++ hay que escribir unos métodos especiales dentro de la clase que queremos insertar en el bloque S-Function para que sean utilizados por Simulink:

static void mdlInitializeSizes(SimStruct *S);

Este método es utilizado por simulink para determiner las cracterísticas del bloque S-Function (número de entradas, salidas, estados, etc.).

static void mdlInitializeSampleTimes(SimStruct *S);

Esta función se usa para establecer el período de muestreo.

static void mdlStart(SimStruct *S);

Este método se llama una vez al principio de la ejecución del modelo en Simulink. Si el modelo tiene estados se inicializarán aquí.

static void mdlOutputs(SimStruct *S, int_T tid);

Dentro de este método es donde calculamos las salidas de nuestro bloque.

static void mdlTerminate(SimStruct *S);

En esta función tenemos que realizar las operaciones necesarias al finalizar la simulación. Eliminar punteros, etc.

10.5.2.1 Construcción de un modelo en Simulink

Construimos un modelo en Simulink que consta, entre otros bloques, de dos bloques S-Function. Uno de ellos encargado de enviar y recibir datos, y otro encargado de tomar las decisiones sobre el control. El funcionamiento es el siguiente, el emisor/receptor de datos le envía a nuestro simulador los datos iniciales, el simulador actualiza el helicóptero según estos datos y le envía la información del helicóptero. El emisor/receptor cuando recibe la información se la pasa al bloque de control, éste trata la información y calcula los valores que deben ser tomados por el helicóptero. Le manda al emisor/receptor los datos sobre las variables de control y el emisor/receptor se los envía al simulador. El simulador actualiza el helicóptero, le envía la información al emisor/receptor y vuelta a empezar.

El modelo en Simulink tiene algunos bloques para mostrar información y además consta de cuatro bloques de memoria que se utilizan para enviar la información al emisor/receptor. Gracias a estos bloques se pueden establecer valores iniciales de las variables de control. El modelo se ha implementado en discreta, ya que los valores no son continuos, sino que se van muestreando. El modelo es el representado en la figura 10.3.

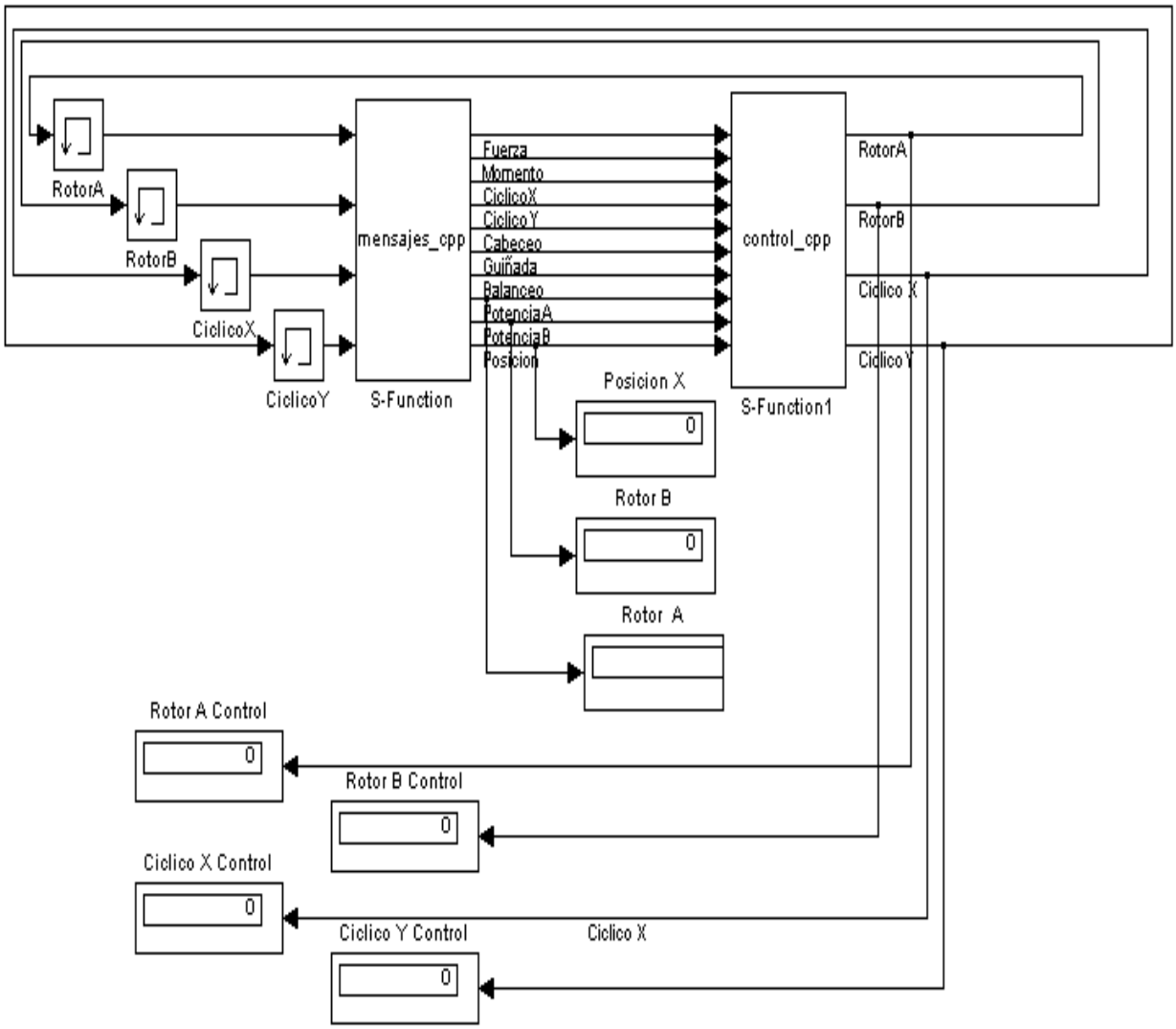


Figura 10.3 Modelo del controlador hecho en Simulink

El bloque de control está basado en una clase de c++ pero la idea es que se implemente con Simulink para obtener un controlador mejor que el que tenemos en este momento.

10.5.2.2 Insertar un bloque S-Function

Para insertar un bloque S-Function en un modelo de Simulink primero hay que arrastrarlo hasta el modelo. Una vez que tenemos el bloque hacemos doble clic para abrir sus propiedades. Lo primero es vincular el bloque a una clase de c++. Para ello escribimos el nombre de la clase implementada en el archivo .cpp que tengamos. Por último hay que indicarle al bloque los parámetros que necesite, en caso de necesitar alguno. Una vez hecho esto ya tenemos el bloque S-Function listo.

10.5.2.3 Compilador de Matlab

Como ya se ha comentado antes, las clases c++ deben tener unos métodos especiales. A parte de esto, los archivos que queremos usar en los bloques S-Function tienen que ser compilados por Matlab. El compilador de Matlab se llama “mex”. Para

ejecutarlo, lo hacemos desde la ventana de comandos de Matlab. La sintaxis para ejecutar mex es la siguiente:

MEX [opcion1 ... opcionN] archivo1 [... archivoN]
[objeto1 ... objetoN] [libreria1 ... libreriaN]

Para visualizar la ayuda se ha de teclear:

mex –h

Para compiler los archivos primero hay que configurar mex, para ello tecleamos el commando:

Mex –setup

Primero nos preguntará si queremos que localice compiladores ya instalados. Le decimos que sí (y). Nos mostrará una lista de compiladores y elegiremos el número de compilador que queremos que utilice mex. A veces un compilador da errores de librerías y hay que probar con otro. Luego confirmamos que la elección es correcta y ya está configurado mex para usarse.

Al compilar es bueno hacerlo con la opción de depuración, para ello se compila tecleando (por ejemplo):

Mex –g mensajes_cpp.cpp

Mex nos informará de posibles errores de compilación y nos indicará la línea dónde están dichos errores, en caso de haberlos.

Al compilar los archivos obtenemos librerías .dll que son las usadas por los modelos de Simulink.

10.5.2.4 Implementación de clases c++ compilables por MEX

Ahora veremos la información a tener en cuenta en la implementación de las clases.

Dentro del método **static void mdlInitializeSizes(SimStruct *S);** hay que especificar los siguientes datos:

El número de parámetros que queremos que tenga el bloque, esto lo hacemos con:

ssSetNumSFcnParams(SimStruct *S, int_T nSFcnParams);

El número de estados continuos y discretos que tiene el bloque:

void ssSetNumContStates(SimStruct *S, int_T n);
void ssSetNumDiscStates(SimStruct *S, int_T n);

El número de entradas y de salidas que tiene el bloque:

boolean_T ssSetNumInputPorts(SimStruct *S, int_T nInputPorts);
boolean_T ssSetNumOutputPorts(SimStruct *S, int_T nInputPorts);

El tamaño de cada entrada y cada salida:

```
void ssSetInputPortWidth(SimStruct *S, int_T port, int_T width);
void ssSetOutputPortWidth(SimStruct *S, int_T port, int_T width);
```

Si las entradas se van a usar para calcular las salidas se deben establecer a cierto las siguientes propiedades:

```
void ssSetInputPortDirectFeedThrough(SimStruct *S, int_T port,
  int_T dirFeed);
void ssSetInputPortRequiredContiguous(SimStruct *S, int_T port,
  int_T flag);
```

Por último, también hay que reservar los elementos que vamos a utilizar del vector de trabajo:

```
void ssSetNumPWork(SimStruct *S, int_T nPWork)
```

Dentro de la function **static void mdlInitializeSampleTimes(SimStruct *S);** hay que establecer el período de muestreo y el offset:

```
void ssSetSampleTime(SimStruct *S, st_index, time_T period);
ssSetOffsetTime(SimStruct *S, st_index, time_T period);
```

Al iniciar el bloque con **mdlStart(SimStruct *S)** hay que crear los objetos que vamos a utilizar en el vector de trabajo. Para ello creamos los objetos en las posiciones adecuadas del vector de trabajo, por ejemplo, si vamos a utilizar dos objetos ObjetoA y ObjetoB irán en las posiciones 0 y 1 (el sitio tiene que ser reservado previamente):

```
ssGetPWork(S)[0] = (void *) new ObjetoA();
ssGetPWork(S)[1] = (void *) new ObjetoB();
```

En este método es dónde creamos el socket en nuestro programa. Construimos un nuevo TCPSocket y lo guardamos en la posición 0 del vector de trabajo.

En el método **mdlOutputs(SimStruct *S, int_T tid)** es dónde calculamos las salidas, para ello primero hay que obtener las entradas necesarias y todas las salidas, esto se hace con los métodos:

```
const real_T *ssGetInputPortRealSignal(SimStruct *S, inputPortIdx)
real_T *ssGetOutputPortRealSignal(SimStruct *S, int_T port)
```

Una vez que tenemos almacenados las salidas y las entradas en las variables adecuadas, le asignamos a las salidas los valores que queremos devolver. Hay que tener en cuenta el ancho de las salidas. Por ejemplo, si tengo una salida con un ancho de dos tendría que hacer:

```
real_T  *y0 = ssGetOutputPortRealSignal(S,0);
y0[0]=valor1;
y0[1]=valor2;
```

En nuestro programa sacamos el socket creado del vector de trabajo para enviar y recibir información a través de él:

```
TCPSocket *c = (TCPSocket *) ssGetPWork(S)[0];
c->send(sendBuffer, sizeof(sendBuffer));
```

Para terminar, en el método **mdlTerminate(SimStruct *S)** tendremos que borrar los objetos creados en el vector de trabajo. Si seguimos con el ejemplo de ObjetoA y ObjetoB tendremos que hacer:

```
ObjetoA* a = (ObjetoA*) ssGetPWork(S)[0];
ObjetoB* b = (ObjetoB*) ssGetPWork(S)[1];
if(a) delete a;
if(b) delete b;
```

En nuestro programa aquí eliminamos el socket que habíamos guardado en el vector de trabajo.

11.DIALES

En esta sección vamos a explicar el funcionamiento y utilidad de todos los diales de los que se ha dotado al helicóptero.

11.1 El HUD

11.1.1Definición informal de HUD

En la mayoría de los videojuegos, sean de el estilo que sean, es necesario el conocimiento de información adicional del estado de la partida que no es posible obtener directa o fácilmente de la escena de juego, esta información puede abarcar cualquier campo, desde las vidas restantes del protagonista o los puntos acumulados, como podría ser en juegos clásicos como el *"Space Invaders",* hasta información sobre la localización del protagonista en un mapa de juego como aparece en juegos más complejos actualmente.

Pues bien, esta información es necesario representarla de alguna manera sin que entorpezca el desarrollo de la partida, y aquí es donde entra el concepto de HUD, por hacer una definición "de andar por casa" un HUD es como si se colocara encima de la pantalla de juego una lamina transparente con esta información pintada en ella .

11.1.2 El HUD en nuestra aplicación

Hemos dicho que la información requerida varía según lo que el programador quiere que conozca el usuario, pues bien, en un simulador de helicópteros (o de cualquier otro vehiculo) lo lógico será que en el HUD aparezca la información de la que puede disponer un piloto real, esto es, los indicadores de la cabina del helicóptero, los que nosotros hemos implementado son los siguientes:

- Indicador de potencia del rotor principal.
- Indicador de combustible.
- Radar tridimensional que indica la posición de posibles objetivos con respecto a la posición del helicóptero.
- Horizonte artificial que indica la orientación del helicóptero con respecto al mundo.
- Esquema de la palanca del cíclico, para tener una idea visual de que posición tiene.
- Indicador del cabeceo, da una idea de la dirección vertical que lleva el helicóptero.
- Altímetro.
- Brújula.
- Gráfico con las velocidades y aceleraciones angulares.
- Mapa con la posición del helicóptero en el plano.
- Además aparece en el HUD un párrafo de texto con diversa información útil para la comprobación del buen funcionamiento del simulador.

Por defecto estos indicadores tienen una posición asignada por nosotros, pero es posible que algunos de estos indicadores ofrezcan información que no interesa al usuario, o que sí que interese, pero no está en el lugar más cómodo para éste. Pues bien, el HUD es configurable a través de un archivo de configuración como veremos más adelante.

La apariencia por defecto del HUD es la que aparece en la siguiente imagen.

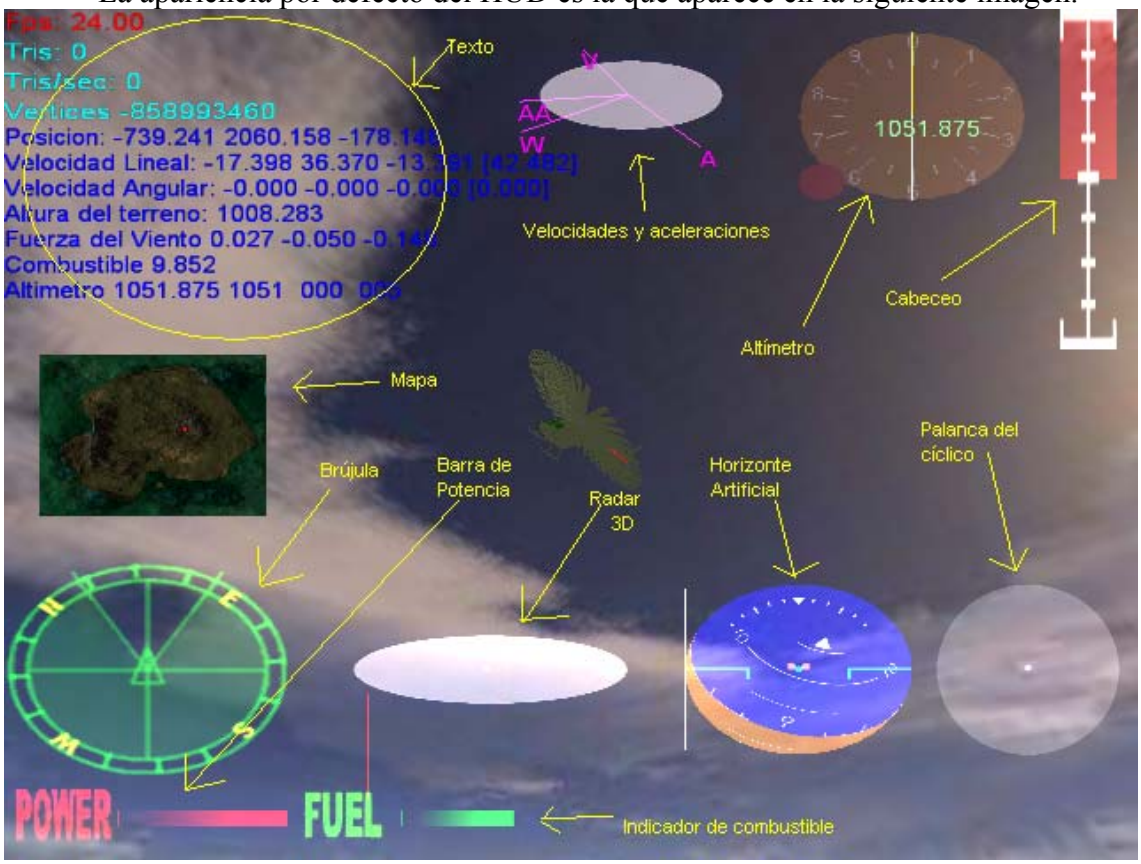


Figura 11.1 Apariencia del HUD

11.1.3 Implementación

La implementación de este HUD se hace a través de la clase *HelicopterGUI*.
Esta clase se vale de la clase *D3DWrapper* (en adelante lo llamaremos raster, ya que es el encargado de rasterizar las imágenes en pantalla), para dibujar los elementos en la pantalla, tiene definida una estructura interna *VertexFomat* de la siguiente manera:

```
struct VertexFormat{
        float x,y,z;
        int color;
        float u,v;
};
```

Los distintos atributos de esta estructura definen lo siguiente:

x,y,z: Coordenadas en pantalla.
color: Color en las coordenadas descritas por x,y,z.
u,v: Normales en ese punto, esta información se usa para colocar la textura.

Para dibujar un elemento en pantalla, el HUD debe hacerse "cliente" del raster al crearse el HUD. En cada llamada al metodo *DrawGUI* de la clase *HelicopterGUI* se establecen en el raster las matrices adecuadas para la correcta visualización de los elementos:

- Tanto la matriz del mundo como la matriz de vista pasan a ser la matriz identidad.
- La matriz de proyección, sin embargo, pasa a ser la matriz:

$$\begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & -2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -1 & 1 & 0 & 1 \end{pmatrix}$$

Una vez establecidas las matrices, se solicita a la clase *DynamicVBManager* un buffer del tamaño adecuado para poder dibujar todos los elementos y va llamando con este buffer y un contador de la posición en la que se tienen que establecer las siguientes coordenadas a los métodos para visualizar cada indicador.

Antes de ver cómo se pinta cada elemento explicaremos brevemente el funcionamiento de las texturas del HUD:

Una textura es un elemento LPDIRECT3DTEXTURE8 de directX, para poder usar una textura debemos declarar un objeto de esta clase, y pedir al raster que cargue en este objeto un archivo de imagen mediante el método *LoadTexture.* En la clase *HelicopterGUI* todas las texturas se cargan en el método *ManageResourceCreate*. Una vez que se sabe que no se va a usar más una textura, se debe eliminar. Esto se consigue con el método *Release* de LPDIRECT3DTEXTURE8, en *HelicopterGUI* todas las texturas activas se liberan mediante el método *ManageResourceDestroy.*

11.2 HORIZONTE ARTIFICIAL

El horizonte artificial es uno de los diales más comunes en los simuladores de avión o helicóptero, debido a que la información que proporciona es muy necesaria para hacerse una idea de cómo estamos orientados en cada momento.

El objetivo principal del horizonte artificial es justamente ese: darnos la orientación del helicóptero con respecto a su posición normal, es decir, con respecto a la posición de cuando está recto, como cuando está parado en el hangar en suelo firme.

Para ello debemos conocer en todo momento la orientación del helicóptero. Esta orientación viene dada por la rotación del eje ligado al cuerpo del helicóptero con respecto a un eje fijo en el suelo. Esto se ilustra en la figura 11.2:

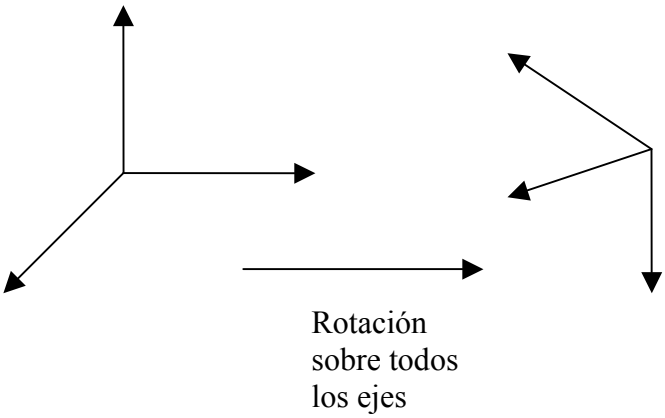


Figura 11.2 rotación de ejes

Esta rotación se suele representar con una matriz de rotación, aunque es más eficiente compactar dicha matriz en un quaternión (véase el apartado de Mejora Gráfica→Orientación de los Objetos).

Vamos a centrarnos en el uso de dicha información para construir nuestro Horizonte Artificial. La pregunta principal es: ¿Cómo consigue el Horizonte Artificial dar una idea de la orientación del helicóptero de forma gráfica? Aunque como veremos hay distintas versiones de Horizonte Artificial posibles, (en 2 Dimensiones o en 3, muy preciso u orientativo), su base consiste en emular el horizonte que vemos desde la cábina del helicóptero (de ahí su nombre). Para representar dicho horizonte no hay más que dividir el cuadrado (o circunferencia o esfera...) horizontalmente en dos: la parte de abajo representa el suelo y la parte de arriba el cielo. De modo que si el eje del helicóptero coincide con el de referencia tendríamos algo como la primera figura, mientras que si por ejemplo el morro del helicóptero está orientado un poco hacia abajo manteniendo su orientación horizontal veríamos algo parecido a lo de la segunda figura (figura 11.3):

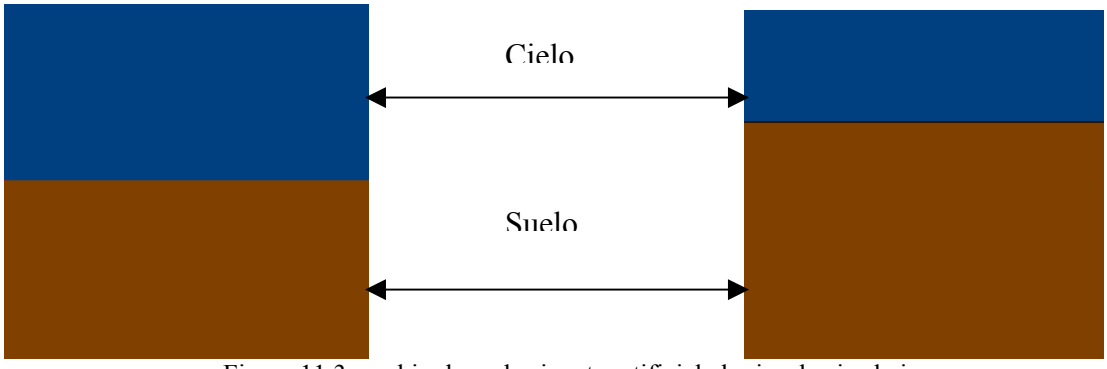


Figura 11.3 cambio de un horizonte artificial al mirar hacia abajo

Esto es lógico, ya que al mirar hacia abajo vemos más tierra y menos cielo. También hay que decir que la rotación sobre el eje Y no se tiene en cuenta, ya que dicha rotación no cambia la fisionomía del suelo y del cielo que estamos viendo. Sólo se tendrá en cuenta la rotación sobre las X (la vertical del morro), que es la ya ilustrada, o la rotación sobre las Z , que nos daría algo como lo mostrado en la figura 11.4:

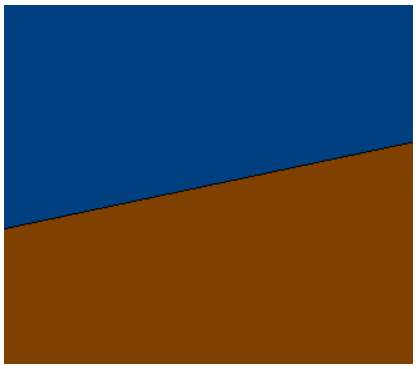
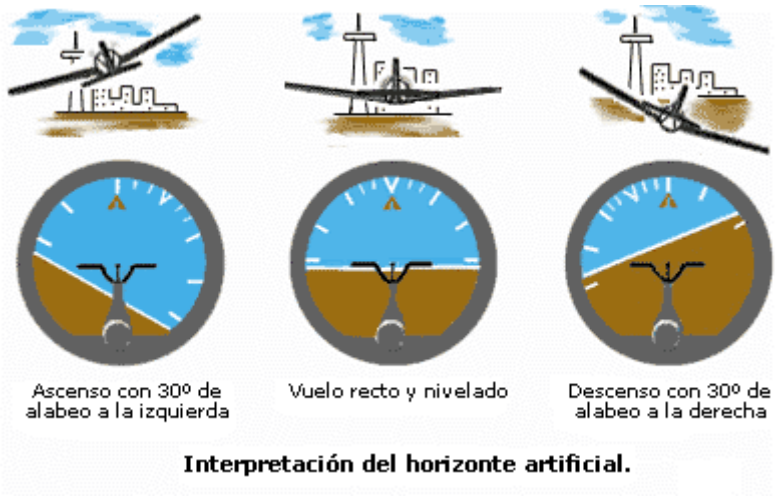


figura 11.4 horizonte artificial al ladearse



Los colores que se escogen para representar cielo o suelo pueden variar, aunque suele ser un color azul o claro para el cielo y oscuro para el suelo. Una vez establecidas se suelen añadir líneas o puntos para indicar los grados de rotación, para orientar sobre la cantidad de giro que se está experimentando y no sólo en que sentido. Para conseguir esto, hay que refinar un poco lo anterior. Lo que se hace es disponer de dos cuadrados superpuestos. El primero es el que ya hemos visto y que cambia según la orientación, mientras que el segundo es un cuadrado fijo que se usa para llevar a cabo las mediciones (figura 11.5):

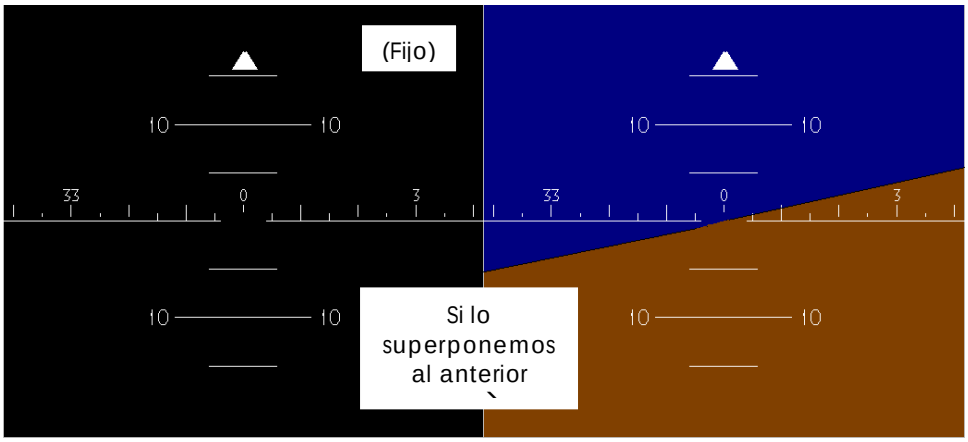


figura 11.5 ejemplo de horizonte artificial con medidas

Vemos que una vez superpuestos nos da información de la cantidad de giro, además de cómo está orientado el helicóptero. En los horizontes artificiales más modernos se suele usar ya una parte de lineas/medición con la parte rotacional y se consigue lo

anterior mediante un sistema más complicado, aunque las bases son estas. Como se verá más adelante (en la parte de implementación), nosotros hemos usado un sistema de este último tipo (la parte rotacional lleva las líneas y números relativas a la medición).

Si miramos los distintos simuladores que hay en el mercado vemos que este dial puede cambiar bastante de uno a otro, ya que como ya hemos dicho, se puede implementar de diversas formas y precisión.

En la figura 11.6 podemos ver 2 ejemplos de estos diales:

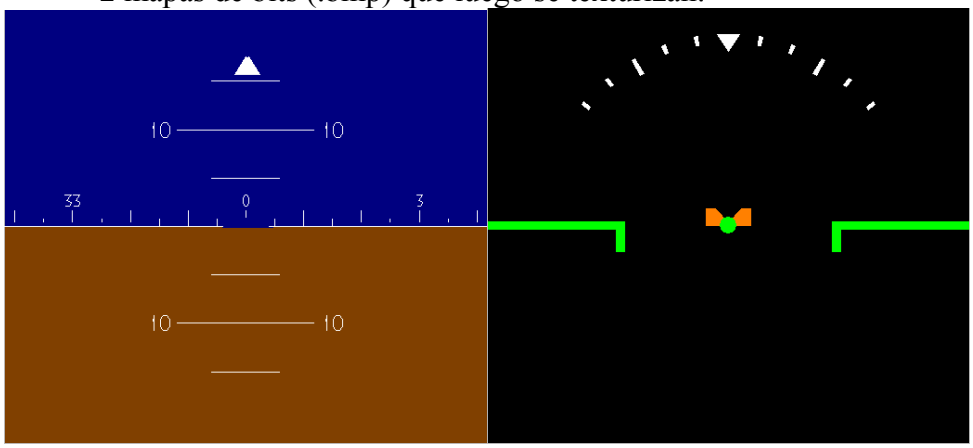
Figura 11.6 Ejemplos de horizontes artificiales

Cómo se puede comprobar todos suelen tener los elementos básicos descritos anteriormente.

11.2.1 Implementación

El horizonte artificial que se ha llevado a cabo en este proyecto es un horizonte artificial basado en una esfera (en lugar de un cuadrado), y usando el esqueleto (parte fija) que proporciona Simulink . Para su implementación se han usado tres componentes gráficos:

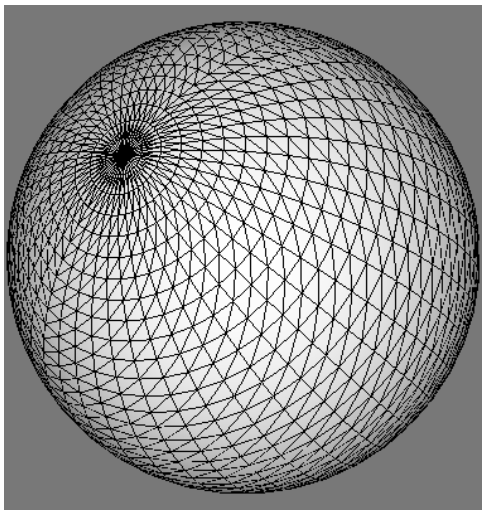
- 2 mapas de bits (.bmp) que luego se texturizan:



Parte rotacional Parte fija

Nótese que la parte fija tiene el fondo negro para que al texturizarla sea trasparente y nos deje ver lo que habrá debajo (la esfera con la parte rotacional).

- Y un archivo .X para la esfera:



Lo que hacemos es primero importar la esfera mediante la función de DirectX **D3DXLoadMeshFromX(...);** que guarda la Malla en una estructura de tipo **LPD3DXMESH**. Al principio cargábamos la malla de la esfera cada vez que se necesitaba, pero retardaba mucho la ejecución y por eso decidimos cargarla sólo una vez en el constructor de la clase HelicopterGUI para ganar en eficiencia.

Luego se cargan todas las texturas necesarias a partir de los .bmp. Una vez hecho esto, definimos un cuadrado tal que contenga exactamente a la esfera (como el de la figura anterior) y en el texturizamos el mapa de bits correspondiente a la parte fija. Esto se consigue mediante el uso de las primitivas directX (después de definir el cuadrado):

raster->SetTexture(0,texturaFija); y
raster->DrawPrimitive(D3DPT_TRIANGLEFAN...);

De esta foma estamos pintando encima de la esfera la parte fija que nos proporcionará la información de cuánto estamos girados.

También hay que sacar la orientación actual del helicóptero (quaternión) llamando a la funcion de **Helicopter getRot();** y rotar la matriz de proyección del rasterizador una cantidad igual a la rotación del eje del helicóptero con respecto a los ejes X y Z, para rotar la esfera proporcionalmente a la rotación del eje del helicóptero. Para hacerlo usamos las facilidades de la clase **Matriz.cpp** para hallar la matriz, y después cambiando la de proyección usando la función **SetProjectionTransform(...);** de **D3Dwrapper** (el rasterizador).

Ahora ya podemos dibujar la esfera (a partir de la malla que hemos cargado), asignándole antes la textura correspondiente a la parte rotacional. Para pintar la malla se usa la función de **LPD3DXMESH drawSubset(...);**

El resultado obtenido se ve en la figura 11.7:

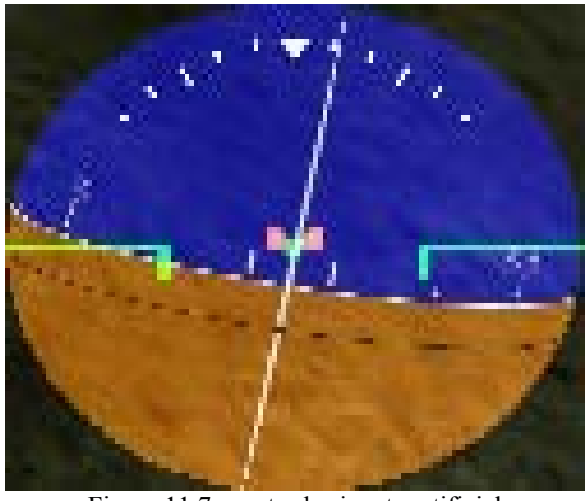


Figura 11.7 nuestro horizonte artificial

El horizonte artificial se implementa en un único método, (como todos los diales). Está en la clase **HelicopterGUI** (también como todos los diales), y tiene como cabecera:

void horizArtif(Dibujable dib,int i,LPDIRECT3DVERTEXBUFFER8 vb,VertexFormat *buf,int &offset);

A este método se le invoca desde el método **DrawGui()** de **HelicopterGUI** (igual que a todos los demás). A **DrawGUI()** se le llama repetidas veces a lo largo de una ejecución para llevar a cabo las funciones de refresco y actualización de todos los diales.

11.3 RADAR 3D

El radar es también una herramienta imprescindible, ya que suele quererse conocer la ubicación de objetos móviles o fijos con respecto a la ubicación actual del helicóptero en todo momento. Este dispositivo realmente no necesita introducción, ya que su uso es muy común y no sólo como instrumento de vuelo, si no que es universalmente utilizado.

El radar implementado en este proyecto es un radar orientado a darnos la ubicación exacta (en 3 Dimensiones) de los objetos de la Escena, con respecto siempre a la posición del helicóptero. Este tipo de radar es más difícil de encontrar que el horizonte artificial en los simuladores del mercado, aunque los más exigentes lo suelen llevar debido a su gran utilidad. Se decidió incorporar al proyecto el radar cuando se añadieron Objetos Dinámicos tales cómo tanques y otros helicópteros, para así poder localizarlos, ya que al ser el mundo tan grande era difícil seguirles el rastro.

¿Qué significa exactamente eso de “en 3D”? Porque todos conocemos el radar en 2 dimensiones típico en muchos juegos de ordenador o incluso en las películas de espionaje, pero ¿cómo es en 3 dimensiones?. Pues bien, la idea es que el radar además de la posición del objeto en cuánto a si está delante/detrás y a la izquierda/derecha, de también información sobre si el objeto en cuestión se encuentra por encima/debajo nuestra. En definitiva, con este tipo de radar sabemos exactamente donde se encuentra el objeto, y gracias a él se puede encontrar muy fácilmente.

La idea básica para construir este tipo de radar es proyectar la información de un típico radar en 2D cómo si estuviésemos volcando este y luego añadirle un componente de altura. Para entender mejor esto se explicará gráficamente en la figura 11.8:

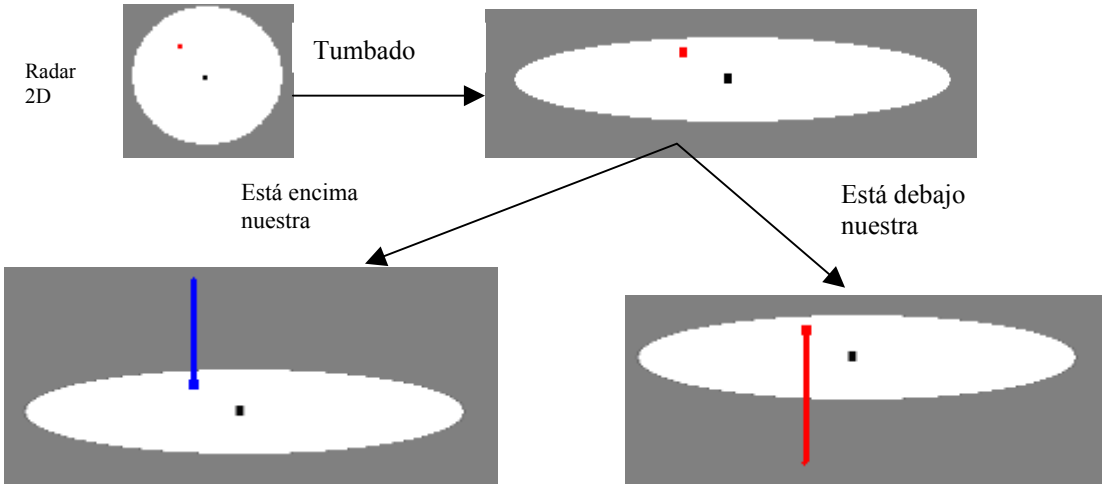


Figura 11.8 uso de un radar 3D

Con esto conseguimos localizar todos los objetos de forma instantánea.

11.3.1 Implementación

En este proyecto se ha implementado un radar 3D muy parecido al de las figuras anteriores. Nos da información tanto de todos los objetos dinámicos presentes en la escena, cómo de los objetos estáticos que hay cerca al helicóptero (en el apartado de Objetos se explica cómo se lleva la lista de Objetos estáticos). Los objetos estáticos los muestra en verde mientras que los dinámicos cambian de color según si están encima nuestra o debajo. Si están encima serán de color amarillo mientras que si están debajo serán de color rojo. El resultado es la figura 11.9:

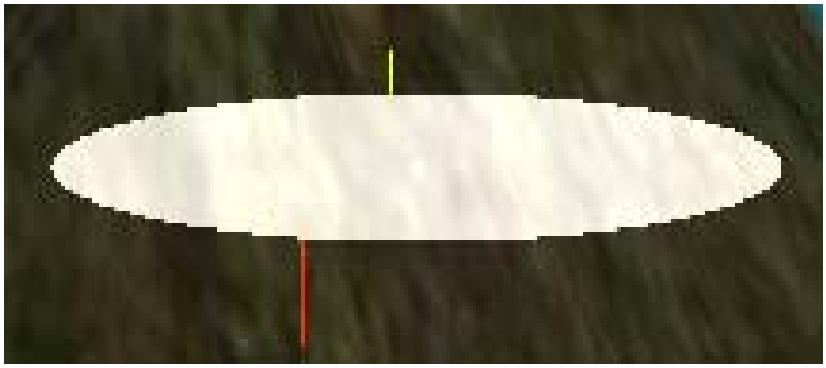


Figura 11.9 nuestro radar 3D con 2 objetos en la escena (uno arriba y otro abajo)

Para hacerlo, primero debemos pintar la elipse. Para ello usamos una textura que no es más que una circunferencia y posteriormente la achatamos para que se convierta en la elipse requerida.Una vez hecho esto pintamos el cuadrado central como punto de referencia.

A continuación, tenemos que ir colocando los objetos existentes según corresponda a cada uno. Para ello hay que tener en cuenta algo muy importante, y es que cuando el helicóptero cambie de orientación (esta vez nos interesan todos los ejes),

todos los objetos tienen que volverse a colocar. Por ejemplo, si tengo un tanque enfrente, y el helicóptero gira a la derecha 90º, ahora dicho tanque estará a mi izquierda, y así con todos los ejes. Esta idea se ve en la figura 11.10.

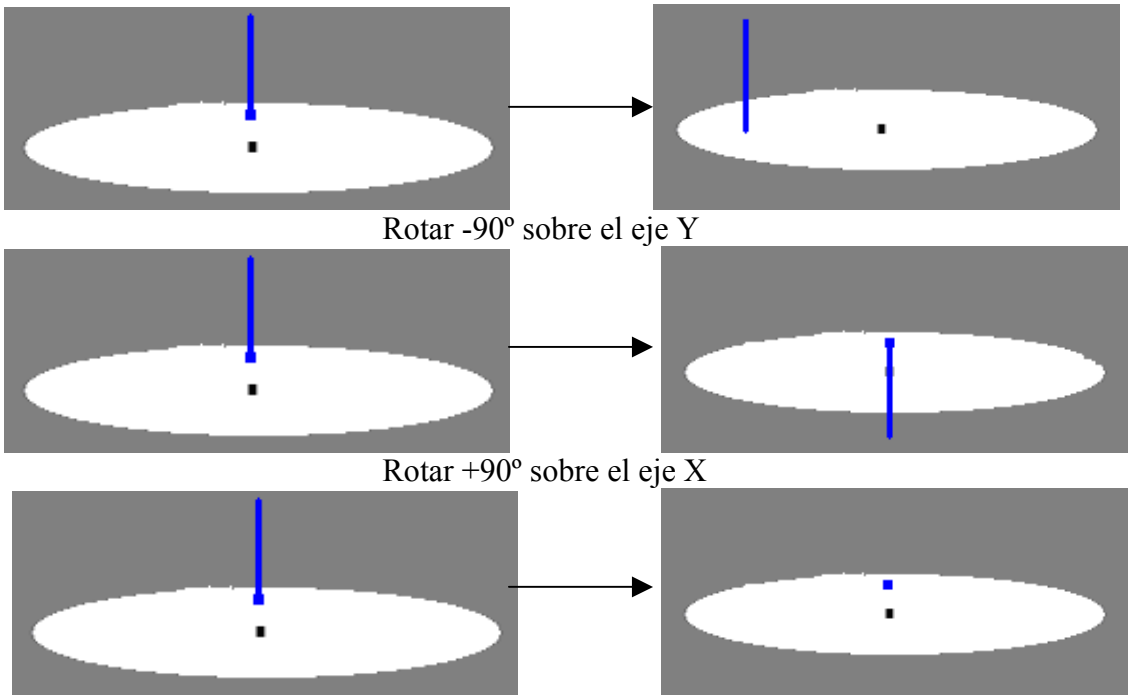


Figura 11.10 funcionamiento del radar 3D

Para conseguir esto hay que hacer una serie de transformaciones sobre las coordenadas de cada objeto, para tener siempre el disco del radar dentro del plano de vista del helicoptero(según como está orientado). Para ello calculamos la posición del objeto con respecto al nuevo sistema de coordenadas trasladado y rotado (posición y orientacion actuales del helicoptero). Esto se hace primero calculando la posición relativa del objeto con respecto del helicóptero y premultiplicando el resultado por la matriz de orientación del helicóptero (en nuestro caso es un quaternión):

Posición objeto i: (xOb,yOb,zOb)
Posición helicóptero: (xH,yH,zH)
Angulos de rotación respecto a cada eje del helicóptero: (rotx, roty, rotz)

Empezamos calculando la posición relativa del Objeto con respecto al Helicóptero :

XOb' = xOb-xH;
YOb' = yOb-yH;
ZOb' = zOb-zH;

Y ahora aunque en nuestro caso no lo hacemos directamente, ya que usamos las facilidades dadas por las clases **Matrix** y **Quat**, explicamos como poner dicha posición relativa en el plano de vista actual del helicóptero. Para ello llevamos a cabo transformaciones sobre cada eje:

x =XOb'
y =YOb'

z =ZOb'

en X: x=x ; y = y*cos(rotx)*sen(rotx) ; z = -y*sen(rotx) +z*cos(rotx)
en Y: x = x*cos(roty) -z*sen(roty) ; y=y ; z = x*sen(roty) +z*cos(roty)
en Z: x = x*cos(rotz) +y*sen(rotz) ; y = -x*sen(rotz) +y*cos(rotz) ; z=z

Y una vez hecho esto **(x,y,z)** me da el punto exacto donde debo situar el objeto en el radar, o lo que es lo mismo, las coordenadas del objeto respecto del eje ligado al cuerpo del helicóptero.

Estas ecuaciones salen del cálculo de transformaciones afines de marcos de coordenadas usadas usualmente en Informática Gráfica. La idea es que representamos el marco de coordenadas actual del helicóptero mediante una matriz de 4x4 de la siguiente forma:

$$R = \begin{bmatrix} R_{11} & R_{12} & R_{13} & 0 \\ R_{21} & R_{22} & R_{23} & 0 \\ R_{31} & R_{32} & R_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

donde (R11, R21 R31) es la posición del eje X con respecto al sistema de referencia.

(R12, R22, R32) es la posición del eje Y.

(R23, R23, R33) es la posición del eje Z.

A partir de esta representación, para hacer las transformaciones a dicho eje se definen las siguientes matrices:

$$T = \begin{bmatrix} 1 & 0 & 0 & X \\ 0 & 1 & 0 & Y \\ 0 & 0 & 1 & Z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matriz de traslación: traslada el eje actual X unidades en el eje X, Y en el eje Y y Z en el eje Z

Matrices de rotación: definen una rotación del marco de coordenadas sobre cada eje con ángulo = Φ.

$$R_{Xrot} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_{Yrot} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_{Zrot} = \begin{bmatrix} \cos\varphi & -\sin\varphi & 0 & 0 \\ \sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Una vez definido esto, para llevar a cabo la transformación deseada basta con descomponerla en transformaciones básicas e ir multiplicando las matrices. Es decir, si tengo (como es nuestro caso) primero que trasladar el marco del objeto hacia el marco del helicóptero y a continuación rotar cada eje con los ángulos rotx, roty y rotz respectivamente, bastará con multiplicar la matriz actual del helicóptero por las cuatro matrices anteriores (fijando el valor adecuado de la traslación y de los ángulos de rotación). Este cálculo nos lo da la ecuación 11.1

Posición del objeto con respecto al helicóptero:

T*Rxroll(rotx)*Ryroll(roty)*Rzroll(rotz) ecuacion 11.1

Lo que nosotros hemos hecho arriba es poner directamente el sistema de ecuaciones resultante, pero es lo mismo.

Esto hay que hacerlo para cada objeto, de modo que debemos recorrer ambas listas de Objetos (Dinámicos y Estáticos , aunque estos últimos se guardan en una tabla Hash →véase el apartado de Objetos), cogiendo las coordenadas de cada objeto, aplicándole la transformación explicada y pintarlo en el radar.

El método de pintado una vez obtenidas las coordenadas transformadas es muy fácil, simplemente consiste en pintar una linea desde el punto **(x,y,z)** al punto **(x,yH,z).**

El radar3D se implementa en un único método. Está en la clase **HelicopterGUI** y tiene como cabecera:

void radar3D(Dibujable dib, int &ps,LPDIRECT3DVERTEXBUFFER8 vb,VertexFormat *buffer,int &offset);

A este método se le invoca desde el método **DrawGui()** de **HelicopterGUI**, igual que en el caso del horizonte artificial.

11.4 VELOCIDADES Y ACELERACIONES

Otro indicador muy útil es el indicador de velocidades y aceleraciones del helicóptero. Es un dial orientado más bien a entender y comprobar la física de un helicóptero. Lo que se pretende es dar una indicación gráfica de la situación de los vectores de velocidades y aceleraciones (lineares y angulares) que "tiran" del helicoptero en todo momento. Se decidió implementar ya que nos pareció útil poder ver en todo momento de forma gráfica "hacia adonde apuntan" las velocidades y aceleraciones. No es muy habitual encontrar este tipo de dial en otros simuladores de vuelo.

11.4.1 Implementacion

Para construir este dial nos hemos basado mucho en el modelo de radar 3D, es decir, la idea gráfica es la misma salvo que ahora se trata de vectores que salen del origen del disco, y no de posiciones de objetos. Por lo tanto se ha usado otra vez la textura con la circunferencia, y se le ha vuelto a aplicar un achatado para que tuviese forma de elipse, aunque como queríamos un disco algo mas pequeño que el del radar se le aplica un escalado posterior también sobre el eje de las X. También se pinta el cuadrado que nos da la referencia de la posición del helicóptero igual que antes. Para dibujar los vectores simplemente accedemos a los valores de las dos velocidades y aceleraciones (lineares y angulares) mediante los métodos de acceso de la clase Helicopter, y pintamos dichos vectores tomando como origen el centro del disco.

El resultado es el de la figura 11.11 (donde se etiqueta con V la velocidad lineal, con W la velocidad angular, con A la aceleración lineal y con AA la aceleración angular):

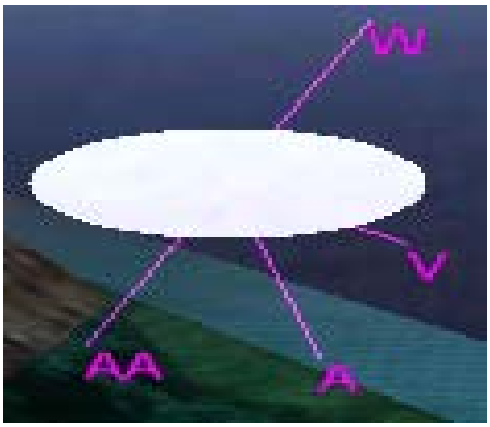


Figura 11.11 nuestro Indicador de V y A

El indicador de velocidades y aceleraciones se implementa en un único método. Está en la clase **HelicopterGUI** y tiene como cabecera:
void drawVyA(Dibujable dib, int &i,LPDIRECT3DVERTEXBUFFER8 vb,VertexFormat *buffer,int &offset);

A este método se le invoca desde el método **DrawGui()** de **HelicopterGUI**, igual que en el caso del resto de diales.

11.5 POSICIÓN EN EL MUNDO

Este dial nos da información de donde nos encontramos en cada momento. Es una herramienta de navegación básica en cualquier tipo de transporte, ya sea en barcos, coches o helicópteros. La infinidad de implementaciones posibles se puede clasificar en base a la precisión o a la parte gráfica (2 dimensiones o 3), así como en base a muchos otros factores. Nosotros, al disponer de un altímetro aparte(explicado en esta misma sección), sólo nos hemos interesado en conocer el punto geográfico sobre el plano del mundo en el que nos movemos, omitiendo información de altura. Es decir, que la representación consiste en un punto sobre el plano del mundo en el que estamos, donde dicho punto nos da la posición actual en la que estamos.

Técnicamente de lo que se trata es de tener una imagen de nuestro mundo desde arriba, como si la hubiese tomado un satélite, y sobre ella proyectar nuestra posición omitiendo información de altura. A medida que nos vamos desplazando por la escena se va actualizando el indicador de posición, indicándonos donde nos encontramos después del desplazamiento.

Este dial es realmente muy intuitivo y de fácil implementación.

11.5.1 Implementación

Antes de nada teníamos que hacernos con una foto del mundo o escenario en el que se mueve nuestro helicóptero. Al principio se pensó en usar directamente el archivo a partir del cual se carga el terreno, pero al añadir lagos de agua estos no se habrían visto, de forma que se desechó la idea. Sin embargo, lo que hicimos fue tomarle una "foto" al terreno desde una posición muy elevada, usando la cámara libre de la que

dispone el proyecto. El escenario visto desde arriba se puede ver en la figura 11.12 (y el punto rojo indicaría nuestra posición):

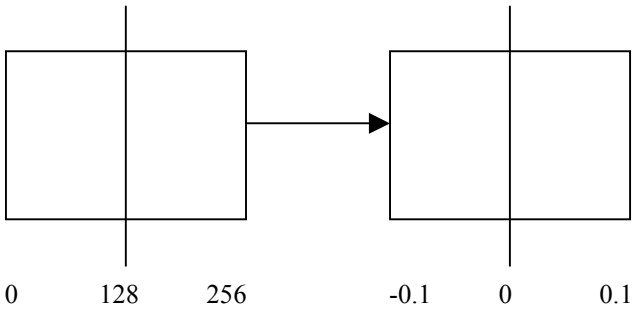
Figura 11.12 nuestra Posición en el Mundo

Una vez logrado esto lo único que hay que hacer (aparte de cargar la imagen y texturizarla sobre un cuadrado predefinido) es pintar un punto que será nuestro indicador de posición. Realmente no es más que pintar un cuadradito rojo, y simplemente ir refrescando su posición, a partir de la posición del helicóptero cada cierto tiempo.

El único problema o cuestión más complicada a tratar ha sido diseñarlo de modo que contemple que el Mundo es realmente "Infinito" (véase la sección de Mundo Infinito). Es decir, este dial tiene que tener en cuenta los saltos que el helicóptero va dando desde una porción de terreno a otra. Lo que se hace es trasformar las coordenadas actuales del helicóptero a coordenadas del terreno, este cambio nos lo proporciona la clase **Terrain** con los métodos **transX(miX) transY(miY) y transZ(miZ),** que devuelven las coordenadas transformadas y donde miX, miY y miZ son las coordenadas actuales del helicóptero. De este modo obtenemos automáticamente las coordenadas sobre el terreno en el que estamos situados y no nos debemos preocupar de los "saltos". La implementación de estos métodos es muy simple, consiste en coger la coordenada que me pasan, ponerla en función del origen del terreno actual y dividirla por un factor de escalado, que en el caso de nuestro terreno es de 20.

Una vez hecho esto, escalamos los valores devueltos en una escala de entre {-0.1→0.1} que es la que usamos para dibujar el cuadrado. Pero antes debemos conocer la longitud del terreno, parámetro accesible a través del método de **Terrain getLength(),** porque sin ello no sabemos con respecto a que escalar.

Para explicarlo mejor, supongamos que el terreno es de 256x256 (longitud = 256):



Luego lo que se hace es **x = (x-128)/(128*10)**

Este dial se implementa en un único método. Está en la clase **HelicopterGUI** y tiene como cabecera:

void drawPosicionMundo(Dibujable dib, int &i,LPDIRECT3DVERTEXBUFFER8 vb,VertexFormat *buffer,int &offset);

A este método se le invoca desde el método DrawGui() de HelicopterGUI, igual que en el caso del resto de diales.

11.6 INDICADOR DE POTENCIA DEL RADAR PRINCIPAL

Este dial ofrece información sobre la potencia que está ejerciendo el rotor principal sobre las palas, lo que da una idea de la velocidad a la que giran éstas y la altura a la que puede elevarse el helicóptero.

11.6.1 Implementación

Hay varias posibilidades para indicar la potencia que está ejerciendo un rotor, una de ellas es la de utilizar un indicador numérico para expresar cuantitativamente esta potencia, otra es representarla mediante una barra que aumenta o disminuye según una cierta escala asociada a la potencia del rotor. Nosotros no tuvimos que tomar una decisión respecto a este tema puesto que ya estaba implementado en la aplicación de la que partimos, sólo hemos tenido que modificarlo.

Pues bien, la implementación de la barra consiste en representar la barra de potencia en rojo y cerca de ella la palabra *"power"* para que no haya dudas acerca de la interpretación que debe darse a éste indicador. La apariencia de esta barra es la que aparece en la figura 11.13:

Figura 11.13 Barra de potencia

El indicador de potencia se dibuja en dos métodos, uno para dibujar la palabra potencia, y otro para dibujar la barra indicadora; estos métodos son, respectivamente, **drawPower** y **drawBarraPower***:*

- **drawPower***:* Éste método utiliza cuatro bloques del buffer para definir un rectángulo que rellena con la textura contenida en texture0, que es una imagen con la palabra "Power"
- **drawBarraPower***:* Éste método utiliza cuatro bloques del buffer para definir un rectángulo que rellena con la textura contenida en texture1, que es una imagen de una barra de color rojo. La longitud horizontal del rectángulo que define depende del valor de la velocidad del rotor principal del helicóptero.

11.7 INDICADOR DE COMBUSTIBLE

Al manejar cualquier tipo de vehiculo, es imprescindible conocer la cantidad de combustible restante en el depósito. Para ello hay diales que lo indican, de una forma u otra.

11.7.1 Implementación

Por intentar hacer un poco homogénea la representación de los indicadores, optamos por implementar el de combustible de la misma manera que el de potencia, así que el dial que indica la cantidad de combustible que le queda al helicóptero se representa como una barra color verde acompañada de la palabra *"fuel"* para identificarla. Al igual que ocurre con la barra de potencia, la barra de combustible aumenta o disminuye de longitud en función del valor del parámetro que indica el combustible del helicóptero. La apariencia de este indicador es la que aparece en la figura 11.14

Figura 11.14 Barra de combustible

El indicador de combustible se dibuja de la misma manera que el de potencia del rotor principal, con dos métodos **drawFuel** y **drawBarraFuel**, que funcionan exactamente de la misma forma que sus equivalentes explicados más arriba.

11.8 PALANCA DEL CÍCLICO

En un helicóptero real, el piloto no tiene más que bajar la mirada para conocer la posición de la palanca que controla el cíclico del helicóptero. Pero en una simulación esta palanca no existe, así que hay que elaborar un método para representarla.

11.8.1 Implementación

Como en la mayoría de las cosas, hay varias opciones, pero la más sencilla es hacer una representación esquemática de la palanca, con un punto simulando el eje de la palanca y un círculo que delimite la posición de este eje, tal como aparece en la figura 11.15

Figura 11.15 Palanca del cíclico

La palanca del cíclico se dibuja en el método **drawCíclico**, que utiliza ocho bloques del buffer para pintarla, cuatro para la palanca y otros cuatro para dibujar un marco circular como esquema de las posiciones en las que puede estar. El marco es un polígono rectangular relleno con la textura contenida en circle0, y la palanca otro polígono relleno con la textura contenida en circle1.
El método añade al buffer primero las coordenadas correspondientes al marco y después dibuja la palanca a partir de la posición del cíclico del helicóptero, convenientemente adaptadas para que se dibuje adecuadamente dentro del marco.

11.9 INDICADOR DEL CABECEO

Otra de las informaciones que conviene poder tener en cuenta a la hora de pilotar un helicóptero es la dirección vertical de éste. Puede parecer una información irrelevante si tienes como referencia el suelo, pero si la altura del helicóptero aumenta lo bastante como para perder de vista el suelo, ya no es tan obvio averiguar en qué dirección te estás moviendo. Para ello existen indicadores que te informan de esto, nosotros al nuestro lo hemos rebautizado como indicador de cabeceo.

11.9.1 Implementación

Nuestro indicador de cabeceo consiste en una barra vertical que crece en el sentido que tenga la componente vertical de la velocidad del helicóptero, y tiene la apariencia que mostramos en la figura 11.16

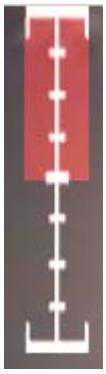


Figura 11.16 Indicador de cabeceo.

El indicador de cabeceo se dibuja en el método **drawCabeceo**, el cual utiliza ocho bloques del buffer para hacerlo, cuatro para un marco con muescas a modo de escala y otros cuatro para pintar una barra que indica hacia qué dirección vertical se dirige el helicóptero. El marco es un polígono relleno con la textura contenida en level, y la barra indicadora del cabeceo es simplemente un polígono coloreado.
La barra indicadora "nace" en la mitad del marco, y crece hacia arriba o hacia abajo en función de la dirección del helicóptero. En términos de código esto significa consultar la componente Y de la velocidad lineal del helicóptero y adaptarla para que pueda representarse gráficamente.

11.10 BRÚJULA

Es imprescindible en cualquier sistema de navegación conocer la dirección en la que te estás moviendo, para averiguarlo el mejor instrumento a utilizar es la brújula, que consiste en una aguja imantada que apunta siempre al norte magnético de la tierra, que se encuentra bastante próximo al norte geográfico.

11.10.1 Implementación

Nuestra brújula consiste en un gráfico de una aguja que siempre apunta al borde superior de la pantalla, y de un círculo con las marcas de los puntos cardinales que gira según la orientación que tenga el helicóptero. Tiene la apariencia que se muestra en la figura 11.17

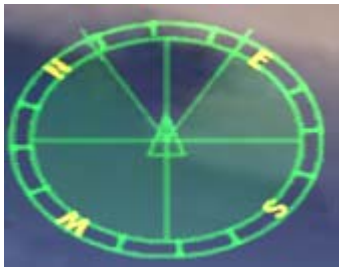


Figura 11.17 Brújula

La brújula se dibuja en el método **drawOrientación**, utiliza ocho bloques para pintar dos elementos, un marco con las señales de Norte, Sur, Este y Oeste y la aguja. Primero se pasa al raster las coordenadas de la aguja, que siempre está fija, y después se calculan las coordenadas del marco basándonos en el ángulo del helicóptero y la posición de la aguja.

Los cálculos para las coordenadas son
Coordenada x = (xHud * cosA - yHud * sinA) + medioX;
Coordenada y = (xHud * sinA + yHud * cosA) + medioY;

Donde (xHud,yHud) es la distancia de las coordenadas de la misma esquina del polígono de la aguja al centro del polígono, determinado por (medioX,medioY) ; mientras que cosA y sinA son,respectivamente, el coseno y el seno del ángulo del helicóptero con respecto a la orientación del mundo.

11. 11 ALTÍMETRO

En cualquier sistema de vuelo se necesita conocer la altitud a la que se está volando, y esto se consigue mediante la inclusión en la cabina de un instrumento que mide la distancia del aparato al suelo: el altímetro.

11.11.1 Implementación

Nuestro altímetro consta de una esfera marcada con números del 0, al 9 y de dos agujas que indican las decenas y las centenas de la altura actual del helicóptero, así como un indicador numérico con el valor de esa altura. Tal y como se puede observar en la figura 11.18

Figura 11.18 Altímetro

El altímetro se dibuja en el método **drawAltímetro** , que utiliza ocho bloques para dibujar una esfera bidimensional con marcas del 0 al 9 y dos agujas que marcan las decenas y las centenas de la distancia del helicóptero al suelo.
En los cuatro primeros bloques se mandan al raster las coordenadas de un polígono en el que pintar la textura almacenada en circle2, los otros cuatro se utilizan para especificar las coordenadas de dos líneas que harán las veces de agujas.
Éste método calcula la posición de las agujas de la siguiente manera:

- En primer lugar obtiene del helicóptero su posición y la altura del terreno en el punto en el que se encuentra.
- De la componente Y de la posición del helicóptero, resta la altura del terreno para obtener la altura del helicóptero con respecto al suelo.
- Obtiene las decenas de esta altura haciendo modulo cien y cogiendo la parte entera de dividir este valor por diez. Para obtener las centenas obtiene el modulo mil de la altura respecto al suelo, le resta el modulo cien calculado antes y obtiene la parte entera de dividir este valor entre cien.
- Las coordenadas de cada manecilla se obtienen mediante trigonometría, al obtenerse los valores para las dos agujas de la misma manera a partir de ahora hablaremos como si de una sola aguja se tratara.
- Dado un valor numérico h, lo multiplicamos por 2π y lo dividimos entre 10, ya que son 10 los valores a los que puede apuntar la aguja, esto nos da el ángulo de la manecilla.
- Calculamos el seno y el coseno de éste ángulo y el radio de la esfera (una de las coordenadas de esta menos el punto medio).
- Las coordenadas de la punta de la aguja serán:

  X = mvX + cos*radio.
  Y = mvY – sin*radio.

  dónde (mvX,mvY) es el punto central de la esfera y , por tanto, las coordenadas de la otra punta de la manecilla.

11.12 TEXTO

El texto que aparece en el HUD se pinta mediante el método **drawText** del raster, en él aparece información sobre la posición y las velocidades del helicóptero, así como la altura del terreno sobre el que se encuentra el helicóptero, la fuerza del viento, el combustible disponible y los cálculos realizados para representar el altímetro. El aspecto de este texto es el que se muestra en la figura 11.19.

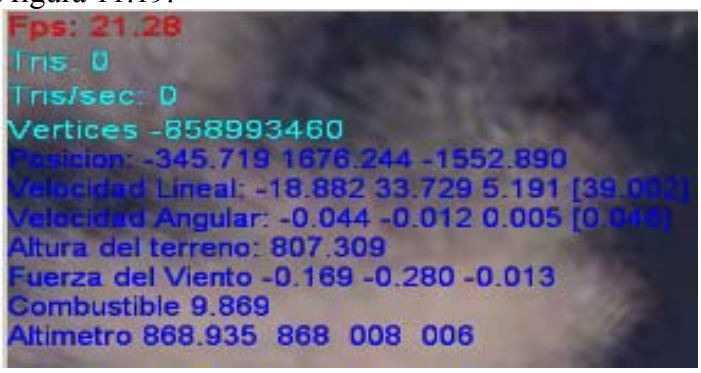


Figura 11.19 Texto del HUD.

11.13 CONFIGURACIÓN DEL HUD

A partir de la versión 1.7 el HUD fue modificándose para hacerlo configurable dinámicamente sin necesidad de recompilar el código del programa, para ello se introduce en la clase **HelicopterGUI** una nueva estructura, la estructura **Dibujable** que está declarada de la siguiente manera:

```
struct Dibujable{
        bool dibujar;
        int numBloques;
        VertexFormat *coordenadas;
};
```

La idea de esta estructura es tener un **Dibujable** asociado a cada uno de los indicadores que pueden aparecer en el HUD; de ésta manera, mediante el atributo dibujar podemos saber si el indicador asociado debe pintarse o no, el número de bloques de buffer que necesita, almacenado en numBloques, y las coordenadas a introducir en estos bloques, almacenadas en el array coordenadas. Se introduce también un array de tamaño fijo de **Dibujable**, que almacena todos los **Dibujable** asociados a cada indicador. Los índices de los indicadores en el array son:

0 – Palabra Power. *(Indicador de potencia del rotor principal)*
1 – Barra Power. *(Indicador de potencia del rotor principal)*
2 – Palanca del cíclico.
3 – Indicador de cabeceo.
4 – Altímetro.
5 – Brújula.
6 – Radar 3D.
7 – Horizonte artificial.
8 – Texto
9 – Palabra Fuel. *(Indicador de combustible)*
10 – Barra Fuel. *(Indicador de combustible)*

11 – Indicador de velocidades y aceleraciones angulares.
12 – Mapa.

¿Cómo se rellenan estos Dibujables? La información para rellenar estas estructuras se obtiene al iniciar el programa del fichero **GUI.cfg** situado en la carpeta BIN/Data/GUI/. En este fichero de texto se encuentra toda la información necesaria para rellenar el array de **Dibujable**, el formato que debe tener este fichero es muy estricto, el más mínimo fallo de fichero hace que no se rellene bien el array. Este formato se basa en el uso de los marcadores % y $. Las restricciones del fichero son las siguientes:

Al comienzo del fichero debe aparecer el símbolo % seguido del número de elementos que contendrá el array.

Después de leer el número de elementos se lee iterativamente la información correspondiente a cada uno, según los índices mencionados más arriba. La información para cada indicador se debe presentar de la siguiente manera:

En primer lugar, debe aparecer el símbolo % seguido de 1 si el indicador debe dibujarse, o 0 en caso de que no deba hacerlo, y el número de bloques de buffer que necesitará el elemento en cuestión. Después de esta "cabecera" deben aparecer tantas líneas de coordenadas como bloques ponga en la cabecera.

Una línea de coordenadas debe empezar por el símbolo % y contener detrás 5 valores numéricos que se almacenarán en los atributos x,y,x,u,v del VertexFormat, en este orden.

Por último, después de todas las coordenadas debe aparecer el símbolo de final $.

La lectura del fichero se realiza en el método **rellenaVisibles** de la clase **HelicopterGUI**. Éste algoritmo recorre el fichero buscando los marcadores %, si no encontrase un marcador donde debiera estar, generaría un error y para evitar excepciones en la ejecución del programa rellenaría los **Dibujable** con unos valores predeterminados.

12. SINGLETONBASE

En esta sección vamos a explicar brevemente el uso de la clase SingletonBase y su utilidad. Esta clase se introdujo al proyecto al introducir el uso del lenguaje de programación Lua para la implementación de los guiones externos (véase la sección de Lua). Su utilidad es que mediante ella se evita tener varias instancias iguales activas de una misma clase, cosa que a veces es necesaria pero que hace perder bastante eficiencia y que desde luego no es nada aconsejable.

Singleton es el nombre que se le da a las clases de las que no queremos tener más de una instancia activa en cada momento. SingletonBase se encarga de asegurar que sólo existirá una instancia de la clase "Clase", dando acceso a esta única instancia a través del método GetInstancia(), como se muestra en el siguiente ejemplo:

Clase & x = Clase::GetInstancia(); ←guardamos en x la única instancia de Clase
x.Metodo1();← ahora ya podemos operar con ella como se haría normalmente

Para convertir una clase "Clase" en Singleton, basta con hacer lo siguiente:

- Heredar públicamente de SingletonBase<Clase>
- Declarar el constructor de Clase, que ha de ser por defecto, como privado
- Declarar la clase SingletonBase<Clase> como amiga, para que ésta pueda acceder al constructor privado de Clase

Dado que no es posible pasar parámetros durante la construcción de los Singletons, si es necesario entonces habrá que añadir un método inicializador que se encargue de configurar por primera vez el Singleton. Ese método inicializador tendrá que ser llamado antes de que ningún cliente haga uso del Singleton. No es posible usar SingletonBase con clases que tengan clases derivadas ni con aquellas que hagan uso de herencia múltiple.

En nuestro proyecto esto se ha incorporado principalmente a la clase **SistemaScripts**, para evitar que cada vez que una clase quiera usar un Guión de Lua (y por lo tanto deba usar alguna función de dicha clase) crease una instancia nueva, ya que a medida que se ha incorporado Lua, el número de clases que necesitan de los servicios de SistemaScripts ha ido aumentado muchísimo. Prácticamente todas las clases nuevas usan Lua, y muchas de las antiguas también, así que con esto se ha ganado muchísimo.

Vemos en código lo que se ha explicado con palabras referente a como convertir a una clase en Singleton (vemos la clase **SistemaScripts**):

```
private:
    /**
     *  Constructor de la clase
     */
    SistemaScripts();

    friend class SingletonBase<SistemaScripts>;
    ... demás parámetros
public:
    ...métodos varios y de Inicialización de variables
```

13. Lua

En esta sección vamos a explicar todo lo referente a la introducción en nuestro proyecto del lenguaje de programación Lua. Hablaremos de qué ventajas ofrece y de porqué se decidió integrar, veremos una descripción básica de su uso y funcionalidad, y acabaremos describiendo dos de los campos en los que más se ha usado: el modelo físico del helicóptero y la introducción de objetos estáticos/dinámicos en el mundo.

13.1 ¿Qué es Lua?

Lua es un lenguaje de programación de alto nivel muy utilizado para el desarrollo de scripts o guiones en juegos. Por guión se entiende toda secuencia de acciones que se debe realizar para llevar a cabo una determinada función. Una forma natural de entenderlo es asignando guiones a comportamientos de agentes (Inteligencia Artificial), o a Inicializaciones de distintos componentes del sistema. Pero no solo se puede usar para esto, si no que al ser un lenguaje completo puede usarse para lo que se desee.

13.2 El porqué de Lua en nuestro proyecto

Decidimos integrar Lua cuándo empezamos a vislumbrar que queríamos ampliar en diversos aspectos la funcionalidad del proyecto y no sólo aportar modificaciones sobre el código ya presente. Empezamos a hablar de la posibilidad de añadir objetos a la escena e incluso de dotarles de algo de inteligencia artificial, de poder cargar de archivo muchas cosas que en ese momento estaban en código C (cómo configuraciones del terreno, inicializaciones varias...), y de poder cambiar el sistema físico para que se adaptase a cualquier modelo de helicóptero...entre otras cosas. Vimos claro que sería muy útil disponer de un lenguaje de scripting, para así poder hacer todos estos cambios de una forma uniforme y rápida. De hecho, al principio, cuándo empezamos a meter objetos estáticos tales como árboles y demás, programamos un cargador de objetos a partir de fichero de producción propia, y nos dimos cuenta del tiempo que llevaba y que incluso con eso no conseguíamos que funcionase como queríamos.

La incorporación de scripts (o guiones) es un requisito fundamental en la programación de cualquier juego o simulador gráfico hoy en día. Esto es debido en gran parte a la cantidad de variables de estado y objetos que es necesario inicializar para poder representar el Mundo y sus componentes, así cómo para la fijación de todos los parámetros de entrada necesarios para llevar a cabo la simulación.

De entre todos los lenguajes de scripting disponibles, escogimos Lua porque era el único que nos resultaba ya algo familiar debido a que ya lo habíamos usado en anteriores proyectos, y que en dichos proyectos anteriores nos había parecido muy útil y fácil de manejar (obviamente si no nos hubiese gustado no lo habríamos vuelto a usar). También hay que decir que el hecho de que no tenga licencia, y que se pueda bajar gratis de internet la última versión así cómo el manual de referencia también ayudó en su selección.

13.3 Ventajas

Disponer de un lenguaje tal como Lua proporciona en nuestro caso 3 grandes ventajas (aunque luego cada una en realidad son varias ventajas juntas):

- Inicializar todos los objetos y variables de estado desde archivo:

Esto permite cambiar lo que se quiera desde fuera del proyecto (desde el archivo que lo contiene) y poder percibir los cambios sin necesidad de volver a compilar el código. Esto, aunque puede parecer una tontería, es muy útil a la hora de probar y cambiar muchos detalles de configuración.

- Tener dos funciones distintas para hacer lo mismo:

  Esto significa poder guardar en un mismo archivo dos funciones distintas para desarrollar una misma operación y poder luego en el código decidir cuál de las dos usar con sólo cambiar una línea. Un buen ejemplo de esto es que cuándo quisimos incorporar un segundo helicóptero y hacer que se desplazase sólo, tuvimos en un principio que crear una clase **Helicopter2**, que realmente no era más que la misma clase que **Helicopter** pero cambiando la función de movimiento para que en lugar de esperar órdenes de teclado se moviese solo. No solo tuvimos que añadir la clase entera, si no que encima tuvimos que englobarlas las dos como subclase de otra (**HelicopterVirtual**) para que en definitiva fuesen dos ejemplos de lo mismo. Con Lua esto se podría haber evitado simplemente definiendo en un mismo script dos funciones de movimiento distintas, y luego asignando a cada instancia de **Helicopter** el uso de una u otra.

  Otro ejemplo de este mismo caso es el modelo físico de helicóptero. Antes el helicóptero tenía asignadas en código C las funciones que simulan su física, y para cambiarlas había que cambiar todo ese código. Ahora, si se disponen de dos modelos distintos de la física, por ejemplo un modelo de Apache y otro de Comanche, basta definirlos en scripts separados y cambiar una línea de código cuándo se quiera usar uno u otro. De hecho esto podría hacerse incluso en tiempo de ejecución bajo petición del usuario...

- Mecanismo automático de lectura y traducción de ficheros:

  Con Lua tenemos automáticamente un compilador y traductor de código a partir de archivo, y esto se puede aprovechar incluso para cosas que inicialmente no necesitaban hacerse en scripts. Un ejemplo de esto es el primer cargador de objetos que se implementó (ya brevemente comentado). Para hacer dicho cargador, además de definir un lenguaje de configuración propio, tuvimos luego que programar los métodos necesarios para leer el archivo y traducir su contenido en acciones. Todo esto se evitó más tarde al añadir Lua, y ahora no solo se puede poner una lista de objetos como en el

cargador inicial, si no que además hemos podido definir funciones tales como **Bosque(x,y,z)** (pone un bosque de árboles en la posición indicada) y cosas así.

No hay, resumiendo, ninguna duda de las ventajas que la introducción de Lua ha supuesto para el proyecto. Pero como hay que ser objetivos, también hay que resaltar uno de sus inconvenientes, y es que, al no haber usado un editor ni compilador particular para Lua, cuando al ejecutar se produce un error en alguno de los scripts, este error puede ser difícil de localizar ya que no se dispone de ninguna herramienta de informe de errores de compilación ni de depuración. Pero siempre se podría usar el entorno de desarrollo LuaIDE sin ningún inconveniente, y resolvería estos inconvenientes.

13.4 Introducción a Lua

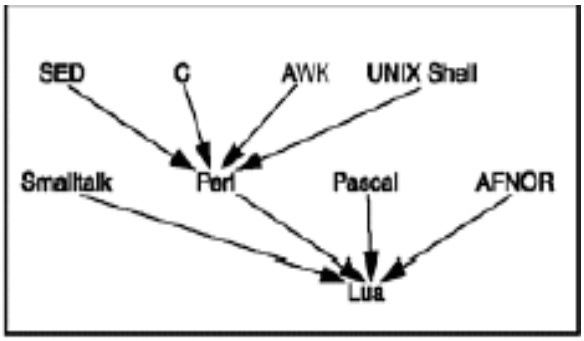


Figura 13.1Arbol genealógico de Lua

Como se puede ver en el árbol, sus predecesores son los lenguajes Smalltalk, Perl, Pascal y AFNOR, y está implementado en C. Es considerado uno de los mejores lenguajes para lograr un rápido y eficiente scripting. Fue desarrollado en 1994 en la Universidad Católica Pontificia de Rio de Janeiro (Brasil). Lua significa Luna en portugués.

Sus características básicas son:

- Tiene una sintaxis simple tipo Pascal.
- Está tipado dinámicamente.
- Gestiona automáticamente la memoria y dispone de un colector de basura.
- Dispone de una sólida descripción de tipos construidos tales como los arrays asociativos.
- Dispone de mecanismos de la Programación Orientada a Objetos tales como clases y herencia.
- Constructores de tipo controlados por el usuario.
- Sus programas son compilados en código-byte y posteriormente interpretados, simulando una máquina virtual

El estudio completo de este lenguaje cae fuera de nuestras intenciones, ya que podría llevar mucho tiempo, y no es lo que más nos interesa. Para conocer más recomendamos que consulten los libros de la Bibliografía (NUMEROS....)

Lo que veremos es cómo hemos integrado dicho lenguaje en nuestro proyecto, una descripción básica del lenguaje y un par de scripts de ejemplo para entender su funcionamiento y estructura.

13.5 Aspectos básicos de Lua

Vemos las funciones matemáticas y algebraicas que proporciona en la figura 13.2:

Function	Lua Command
Add	+
Subtract	-
Multiply	*
Divide	/
Equal (assignment)	=
Equal To	==
Less Than	<
Greater Than	>
Logical NOT or Not Equal To	not
Logical AND	and
Logical OR	or
Square Root	sqrt
Exponent	exp
Absolute Value	abs
Basic Sin	sin
Cosin	cos
Tangent	tan
Logarithm	log
Truncate/Round	round
Floor	floor
Ceiling	ceil
Power	^

Figura 13.2 funciones proporcionadas por Lua

Comentarios (--): --Esto es un comentario en Lua

Variables: Los tipos básicos son: Booleanos, Reales, Enteros, Strings y NIL (sin valor). Una de las características de las variables de Lua es que no es necesario declarar su tipo la primera vez que se usa, lo infiere directamente el compilador. De modo que directamente hacemos X = 'Hola' y reconoce que X es de tipo String.

Además, Lua dispone de todas las instrucciones de control típicas de un lenguaje de programación, como los IF, WHILE, FOR...Así como estructuras de datos construidos como los Arrays, Estructuras...

Todos los métodos que hemos implementado en Lua han sido como funciones, las declaraciones de Funciones en Lua son así:

```
function Ejemplo(param1,param2)
...código
end
```

Todos los scripts que hemos realizado están dentro de la carpeta BIN/scripts, se han editado con el Bloc de Notas, y se guardan como archivos LUA (.lua).

Para poder usar Lua desde nuestro código es necesario añadir al proyecto las librerias y archivos de cabecera que se encuentran en la carpeta LUA.

13.6 Interacción y uso desde C y viceversa

Para la interacción entre nuestro proyecto (código C) y Lua y viceversa hemos diseñado dos clases que por si solas representan un "Sistema" entero de scripts. Son las clases **SistemaScripts** y **ScriptsModelo.** La segunda define la interacción con todas las funciones de Lua referentes a la implementación del modelo físico y la veremos al hablar de dicho modelo, mientras que la primera se encarga de todas las demás funciones y es la que tomaremos como ejemplo para explicar la interacción de Lua y nuestro proyecto.

La clase **SistemaScripts** ofrece la posibilidad de llamar a un script y ejecutar partes del programa cuyas exportaciones se han realizado previamente. Por ejemplo, si tengo el siguiente guión Lua en un archivo (Escena.lua):

```
function CargarEscena(pEscena,pRaster,pTerrain)
        LoadFlare(pEscena,pRaster,-0.813, 0.399, 0.424,pTerrain);
        LoadUnderWater(pEscena,pRaster,300,"texture/water.jpg");--aleatorio
End
```

Y quiero llamar a dicho guion desde mi clase, lo que haría sería:

```
Sc.Cargar("Escena.lua");
Sc.EjecutarFuncion ("CargarEscena",TS_VOID,this, TS_VOID,raster,
TS_VOID,terrain));
```

Donde Sc es la instancia de **SistemaScripts**. Cómo se puede observar, en el paso de parámetros hay que especificar antes de que tipo es cada uno. TS_VOID, en este caso, significa que es un puntero a un Objeto. Este es el sistema de llamada que se repite infinidad de veces a lo largo de todas las clases de nuestro proyecto, siempre que se quiere llamar a una función de algún script. Nótese, además, que a su vez hay una interacción inversa, ya que los métodos a los que se llama desde la función del script son funciones de nuestro proyecto (funciones C). Vamos a ver cómo se consigue establecer está comunicación tan simple.

La clase **SistemaScripts** sólo tiene un atributo, la pila de comunicación con Lua, que es de tipo puntero a **lua_State** (en Lua todo va con pilas). Además tiene como métodos fundamentales **IniciarScripts(), Cargar(),** y **EjecutarFuncion()** entre otros.

El método **Cargar(),** cómo se ha visto en el ejemplo, sirve para cargar un script (archivo LUA) determinado. Lo que hace es un **lua_dofile(pilaLua,ruta)**, que carga en la pila el script indicado en ruta.

El método **EjecutarFuncion()** se encarga de llevar a cabo la llamada a una determinada función definida dentro del guión que ha sido cargado por último. Para ello primero apila en la pila de Lua el nombre de la función a ejecutar con **lua_getglobal(pilaLua, NombreFuncion)**, luego apila todos los parámetros y una vez acabado hace un **lua_call(pilaLua,n,0)** donde n es el numero de parámetros. Lo que hará Lua será ir desapilando uno a uno todos los parámetros y asignarle a cada uno una variable del tipo que corresponde, y con ellos ejecutar la función que queda en la cima de la pila.

Mientras que los dos métodos anteriores corren prácticamente a cargo de Lua, el método **IniciarScripts(),** al que se le llama únicamente una vez (en el constructor), tiene que poner a disposición de Lua todas las funciones de C que querramos invocar después (desde fuera), esto nos obliga a definir para cada función C su correspondencia a Lua. Para explicar esto vamos a ver como hemos puesto la función C usada arriba **LoadFlare(pEscena,pRaster,-0.813, 0.399, 0.424,pTerrain);** a disposición de Lua: Antes de nada definimos dicha función de la siguiente forma:

```
FUNCION(load_flare)
        void* pTerrain = ARGUMENTO_OBJETO;
        float z = ARGUMENTO_REAL;
        float y = ARGUMENTO_REAL;
        float x = ARGUMENTO_REAL;
        void * pRaster =  ARGUMENTO_OBJETO;
        void * puntero=  ARGUMENTO_OBJETO;
        Flare* luz = new Flare((D3DWrapper *)pRaster,(Node*)puntero);
        Point3 lpos(x,y,z);
        luz->setPos(lpos);
        luz->setTerrain((Terrain*)pTerrain);
        ((Node *)puntero)->AddChild(luz);
FIN_FUNCION
```

Cómo se puede ver, no es más que la implementación de lo que se quiere hacer directamente en código C, salvo por las macros usadas, que no son más que las llamadas necesarias a Lua para que gestione la pila al producirse la llamada (desapilado de los parámetros...):

FUNCION(arg)→ **int arg(lua_State * guion){**

ARGUMENTO_OBJETO→ **(void *)lua_touserdata(guion, lua_gettop(guion));lua_pop(guion, 1)**

ARGUMENTO_REAL→ **lua_tonumber(guion, lua_gettop(guion));lua_pop(guion, 1)**

FIN_FUNCION→ **return 1;}**

Esto lo tenemos que hacer para todas las funciones que deban llamarse desde Lua para cambiar cosas del código C. Cómo ha sido necesario definir muchísimas de estas funciones, se han organizado por clases (primero las funciones necesarias para la gestión de la clase **Helicopter**, luego todas las de **Terrain** y así sucesivamente). Estas funciones se hacen al principio de la clase sin estar dentro de ningún método.

Una vez hecho esto, dentro de **IniciarScripts()** (también una a una) indicamos a Lua que la funcion **LoadFlare(...)** que necesita se corresponde con la función **load_flare** de C (la que acabamos de definir), mediante el uso de otra macro:

DEFINIR_FUNCION("LoadFlare", load_flare);

DEFINIR_FUNCION(arg1, arg2)→ **lua_pushcclosure(m_Lua, arg2, 0);lua_setglobal(m_Lua, arg1)**

Con todo esto ya se puede producir la comunicación descrita al principio.

Se han explicado las bases del funcionamiento de la interacción Lua-C en una llamada normal, pero la clase **SistemaScripts** realmente dispone de más métodos de ejecución a parte del de **EjecutarFunción()**. Algunos de ellos para leer de archivo strings o enteros, y otros para permitir usar funciones que no sólo cambien cosas de las clase de C, si no que también devuelvan valores (Booleanos, Enteros...) . No se explicará todo, pero en el código viene todo muy bien comentado para quién quiera usar estas funciones.

Ahora que sabemos usar Lua vamos a ver cómo lo hemos usado para los Objetos y después para el modelo físico de helicóptero.

13.7 Ejemplo 1: Objetos

Al principio simplemente quisimos introducir objetos estáticos tales como árboles, casas y otros para que el mundo no fuese simplemente un paisaje plano, si no que tuviese también los elementos normales de cualquier escenario de vuelo real. Así que se diseñó la clase **Objeto**, para que cada objeto pudiese definirse como una instancia de esta clase.

Para su diseño se tomó como referencia la clase **Helicopter** (al fin y al cabo el helicóptero en si no es más que un objeto), pero quitando de esta clase toda la parte referente a la física y simulación de vuelo. Es decir, se quedó todo lo referente a cargar desde un archivo .X el objeto, e implantarlo gráficamente en la escena en una posición determinada, y se quitó toda la parte de actualización de su posición en función de las fuerzas que actúan sobre el, ya que los objetos estáticos una vez implantados ya no se mueven ni deben sufrir cambios. La clase **Objeto** hereda por una parte de la clase **Node** para pintar una malla de un objeto en pantalla y por otro lado hereda de una ecuación diferencial para integrar el conjunto de fuerzas que actúan sobre el objeto e integrarlas con el integrador adecuado (aunque en los estáticos no se usa).

Como se ve el diseño de esta clase fue por lo tanto inmediato, y nos resultó muy fácil poder incorporar los objetos. Para hacerse sólo necesitábamos un .X. Dichos archivos han sido en su mayor parte cogidos de internet u otros juegos (véase la sección de DirectX→Archivos X).

Una vez que conseguimos cargar un objeto aislado sin problemas, surgió la necesidad de poder cargar varios objetos a la vez para poder así dar más realidad al escenario (un árbol sólo no sirve de mucho, se necesita un bosque de x árboles). Y aunque al principio se cargaban uno a uno en el constructor de la clase **Main**, no se podía dejar así, porque era demasiado farragoso. Se implementó un cargador de objetos (que ya hemos brevemente mencionado), para guardar en un archivo la lista de objetos con sus nombres y posiciones. El archivo de configuración era algo así:

“Arbol1” “data/pino.x” %0 %1100 %0
“Casa1” “data/house.x” %10 %1105 %12
....

El primer valor era el nombre del objeto, el segundo el archivo que contenía su malla, y las tres últimas su posición (x,y,z). Y esto debía hacerse uno a uno para todos los objetos. Para leer de este archivo y cargar uno a uno los objetos se tuvo que diseñar un analizador de dicho archivo que fuese leyendo todos los valores, creando los objetos y añadirlos a la escena. Este proceder habría sido imposible de mantener con el número de objetos actuales. Aquí es donde entra Lua.

Una vez incorporado Lua al proyecto, decidimos dejar de usar por completo esta forma de carga de objetos, y usar en su lugar scripts de Lua. De esta forma no solo nos evitamos tener que diseñar y mantener un analizador del archivo de configuración, si no que ya no es necesario definir uno a uno todos los objetos, ya que se pueden usar las instrucciones facilitadas por Lua. Por ejemplo, si antes queríamos un bosque cuadrado de 10x10, teníamos que poner en el archivo de configuración 100 entradas, cada una referente a un árbol, y además calcular nosotros su posición para formar entre todos un bosque compacto. Con Lua, sin embargo, simplemente se define una función Lua **Bosque**:

```
function
Bosque(pRASTER,pTERRENO,pLISTA,pGRAFO,filas,columnas,xIni,zIni)
        for i=1,filas do
                        for j=1,columnas do
                                PonObjeto(pLISTA,pGRAFO, zIni+j*10, 0,xIni+i*10,
"Data/pino.x", 0.1, 10, pTERRENO,pRASTER);
                        end
        end
end
```

y dándole en filas y columnas el numero de árboles nos crea automáticamente el Bosque. Ha quedado claro el uso y ventaja de Lua para la carga de objetos. Pero aún no hemos acabado de describir la incorporación de objetos.

Hasta ahora hemos hablado sólo de objetos estáticos, objetos que carecen de física o de inteligencia porque no se mueven. Pero más tarde incorporamos también objetos dinámicos, tales cómo otros helicópteros o tanques...Estos objetos sí que necesitan seguir un modelo físico según su naturaleza (el tanque no puede volar, el helicóptero si), y además para que interactúen con nuestro helicóptero deben ser dotados de un cierto control automático o inteligencia. Para estos propósitos se diseñó la clase **ObjetoDin.**

Igual que la clase **Objeto**, la clase **ObjetoDin** hereda de **Node** y de una ecuación diferencial para el cálculo de sus ecuaciones físicas. Además de todas las funciones ya presentes en la clase **Objeto**, a esta clase se le han incorporado muchas funciones adicionales, necesarias para que el objeto se pueda mover e interactuar con el resto del escenario.

- Ecuaciones físicas: En este proyecto se han implementado dos tipos distintos de modelos físicos. Uno es el modelo de un helicóptero, y se usa tanto para el helicóptero principal como para los auxiliares añadidos como objetos dinámicos. El segundo es un modelo de vehículo sobre ruedas, como coches o tanques, usado en este apartado. Ambos modelos están hechos en Lua y se verán más adelante. De momento sólo nos interesa saber que para dotar a un **ObjetoDin** de física basta cargar el modelo deseado de Lua. Así de sencillo.
- Colisiones: Ya que el objeto se mueve, puede colisionar con todo componente de la escena (véase la sección de colisiones). Esto se ha implementado igual que en la clase **Helicopter**.
- Mejoras gráficas: se ha dotado a estos objetos de la posibilidad de disparar proyectiles (lógico en un tanque), y cómo también es lógico, al colisionar con algún elemento del escenario pueden arder o explotar, por lo que se usa el

sistema de partículas. Todo esto se ha implementado igual que para el helicóptero principal, y se explica en la sección de Mejoras Gráficas.

- Control o IA: Si no dotamos a estos objetos de capacidad para desplazarse y tomar decisiones solos, no serían dinámicos ya que no se moverían. Por falta de tiempo sólo se han podido poner ciclos muy básicos de control y sin ser en código Lua, si no en código C directamente. Por ejemplo el tanque lo único que hace es orientarse de forma que tenga a tiro al helicóptero y luego empieza a disparar proyectiles para intentar destruirlo. Sin embargo con algo más de tiempo habría sido muy fácil poner lo mismo en Lua, y a continuación refinar un poco la inteligencia.

Por último decir que todos los objetos que se añaden a la escena, deben ser guardados en estructuras de datos auxiliares para que el helicóptero se percate de su presencia y pueda colisionar con ellos (se explica en la sección de colisiones), y también para que el Radar 3D de cuenta de ellos (se explica en la sección de Diales).

13.8 Ejemplo 2: Modelo físico

El proyecto anterior sobre el que se basa éste, ya tenía implementado una simulación de la física del helicóptero, pero esta física era realmente muy básica y se quiso mejorar. Lo que había (véase la documentación del proyecto anterior), era lo mínimo necesario para que pareciese real y para simular el movimiento del helicóptero. Es decir, estaba implementado todo mediante ecuaciones diferenciales y el método de Euler que las resuelve (método numérico) para hallar todas las fuerzas y aceleraciones, además de las ecuaciones físicas de sólidos rígidos para simular el cuerpo del helicóptero en sí. Pero de ahí a simular realmente con todas sus ecuaciones la física de un helicóptero, había mucho camino. Desde el principio de este proyecto se sabía que unas de las mayores prioridades era cambiar esto. Queríamos que nuestro helicóptero siguiese de verdad un modelo físico lo más real posible.

Pero el modelo físico de un helicóptero es algo realmente muy complicado de simular y de entender, y si hubiésemos tenido que hacerlo desde 0 nos habría llevado muchísimo tiempo. Por eso se ha cogido como referencia el trabajo "MODELACIÓN DEL SISTEMA NO LINEAL DE UN HELICÓPTERO" (véase Bibliografía Libro numº[15]), llevado a cabo el año pasado en esta misma Universidad. A decir verdad, se tenía claro desde el principio que se dotaría al helicóptero de un modelo físico completo basándose en dicho trabajo, de no existir no habría sido tan sencillo.

En dicho trabajo se crea un modelo operativo del sistema del helicóptero a escala maqueta en el ordenador; con el cual poder realizar simulaciones. Se basa en un modelo matemático en ecuaciones diferenciales extraídas de la física que gobierna los diferentes aspectos a tener en cuenta, e incluso tiene en cuenta posibles perturbaciones atmosféricas. Después, en base a dicho modelo matemático, construye el modelo en un entorno MATLAB Simulink.

Lo que se ha hecho para poder basarnos en dicho trabajo, es primero leerlo y entenderlo, y después adaptar lo que se hace para Simulink a código C. Además, una vez conseguido esto, interesaba que dicho modelo pudiese cargarse desde archivo para poder más adelante cambiar el modelo fácilmente o incluso poder dotar a distintos helicópteros de distintos modelos físicos. En esto último es dónde entra Lua.

Lo que se ha hecho es hacer un sistema que lee las ecuaciones físicas desde un archivo. Lo primero que hacemos es leer las constantes que van a afectar al objeto, en el

caso del trabajo del año pasado sería un helicóptero, pero pueden ser ecuaciones físicas de cualquier objeto. Estas constantes son:

FricLineal	fricción del aire a la velocidad lineal
FricAngular	fricción del aire a la velocidad angular
Rozamiento	rozamiento contra el suelo
Viento	indica si debe ser afectado por el viento
Proyectil	nombre del archivo con las ecuaciones físicas de los proyectiles
Explosion	Habilitar/deshabilitar las explosiones y el fuego

Estas constantes se leen una única vez en el constructor del objeto. Luego, el archivo contiene una serie de funciones que se llaman por medio de Lua para calcular distintos aspectos del modelo físico. Dichas ecuaciones se leen y se ejecutan cada vez que hay que actualizar el objeto.

Estas funciones las vamos a ver según su utilidad. Las primeras que vamos a ver son las que calculan fuerzas que se transforman a coordenadas del objeto. Veamos estas funciones :

CalcularFuerzaA(vector,cyclicX,cyclicY,potA,potB,masa)
Esta función calcula la fuerza ejercida por el rotor principal. Recibe como parámetros la posición del cíclico, la potencia de los rotores y la masa del objeto. Utiliza el parámetro vector para devolver en él el resultado.

CalcularFuerzaB(vector,cyclicX,cyclicY,potA,potB,masa)
Calcula la fuerza ejercida por el rotor trasero. Los parámetros son iguales que en el caso anterior.

ModificarCentroA(centro,x,y,z)
Recibe un vector que apunta hacia la dirección (0,0,1) en coordenadas del objeto. Devuelve en el parámetro centro la posición donde queremos que esté el centro de masas que se usará para calcular el momento del rotor principal.

ModificarCentroB(centro,x,y,z)
Igual que el método anterior pero para el rotor trasero.

CalcularMomentoA(vector,fuerzaA,centro,cyclicX,cyclicY,potA,potB)
Calcula el momento del rotor principal y lo devuelve en el parámetro vector. Recibe la fuerza del rotor principal, el centro de masas, la posición del cíclico y las potencias de los dos rotores.

CalcularMomentoB(vector,fuerzaB,centro,cyclicX,cyclicY,potA,potB)
Igual que la función anterior pero recibe la fuerza del rotor trasero y el centro de masas a usar del rotor trasero.

La siguiente función calcula las fuerzas que no se transforman a coordenadas del objeto :

FuerzaExterna(vector,cyclicX,cyclicY,potA,potB,masa)
Recibe la posición del cíclico, las potencias de los rotores y la masa del objeto, devuelve la fuerza en el parámetro vector.
Por ejemplo, en esta función es donde se calcula la fuerza que representa el peso del objeto.

Las siguientes funciones calculan las derivadas que se usan para actualizar la posición, el momento lineal, el momento angular y la orientación del objeto.
DerivadaPos(vector,cyclicX,cyclicY,potA,potB,masa,velX,velY,velZ)
Actualiza la posición delobjeto, los últimos tres parámetros indican la velocidad del objeto.
DerivadaLineal(vector,cyclicX,cyclicY,potA,potB,masa,fX,fY,fZ)
Actualiza el momento lineal. Los últimos tres parámetros indican la fuerza del objeto.
DerivadaAngular(vector,cyclicX,cyclicY,potA,potB,masa,tX,tY,tZ)
Actualiza el momento angular. Los últimos tres parámetros indican el momento del objeto.
DerivadaQuat(quat,cyclicX,cyclicY,potA,potB,masa,rotX,rotY,rotZ,rotW,wX,wY,wZ)
Actualiza el quternión que representa la orientación del objeto. Recibe las coordenadas del quaternion del objeto en este momento y la velocidad angular. Devuelve el nuevo quaternion en el parámetro quat.

La clase que produce la interacción con el modelo físico en Lua es la clase **ScriptsModelo**. Esta clase funciona igual que la clase **SistemaScipts** ya explicada, salvo por las funciones que se definen para que Lua las use, funciones todas orientadas a construir el modelo físico. El helicóptero tendrá acceso a esta clase para usar todas las funciones de Lua que sirven para construir el modelo. El paso de C a Lua fue inmediato porque sólo hubo que definir las funciones necesarias y por el resto copiar la clase **SistemaScripts**. La forma de llamar a dichas funciones desde **Helicopter** es también idéntica al resto de funciones Lua y por eso se hizo también de inmediato.

14. INSTALADOR

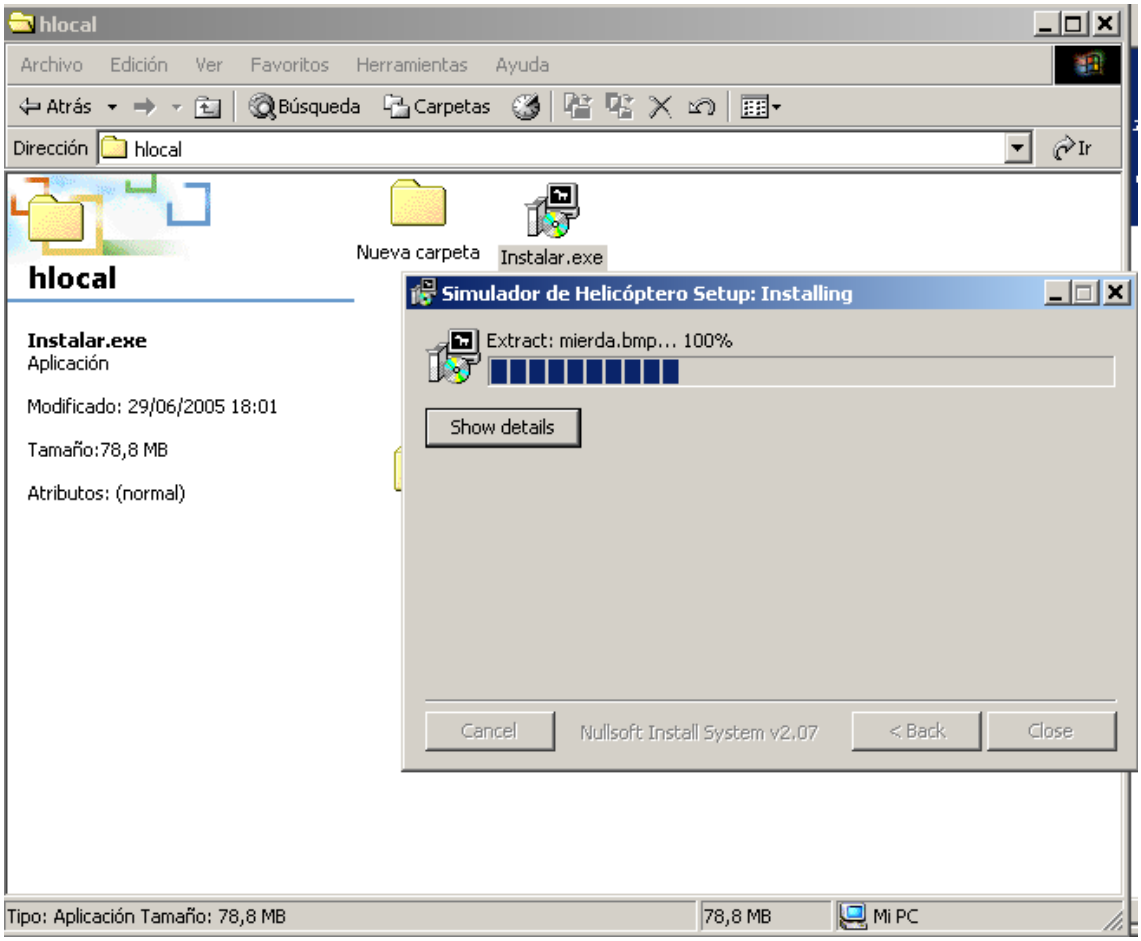


Figura 14.1 Lo que se ve al lanzar nuestro Instalador

Si copiamos sin más los archivos del proyecto a un ordenador determinado, el proyecto muy probablemente no podría ejecutarse, ya que necesita que previamente se hayan añadido las librerías de DirectX, Lua y Fmod al sistema. Para evitar este problema, se ha desarrollado un instalador de forma que el proyecto pueda probarse en cualquier plataforma que lleve instalado Windows.

Este Instalador se ha programado con el sistema de scripts NSIS.

Funciona como cualquier otro instalador existente para programas de Windows. Basta hacer doble clic en el archivo **Instalar.exe**, y automáticamente realiza todas las copias de archivos e instalaciones en el sistema necesarias para que una vez finalizado se pueda lanzar el ejecutable del proyecto desde dicha plataforma. Entre estas acciones se encuentra el lanzamiento automático del instalador de DirectX. Además también puede añadir el programa a la lista de programas de la barra de programas de Windows, para que sea más fácil tener acceso al proyecto en todo momento. Obviamente también cabe la posibilidad de desinstalar lo instalado ejecutando el archivo **Uninstall.exe**.

Para hacerlo se consultaron los tutoriales disponibles en el mismo programa, así como otros encontrados por internet.

15. POSIBLES MEJORAS

15.1 INTELIGENCIA ARTIFICIAL

El proyecto queda preparado para facilitar la inclusión de algoritmos de inteligencia artificial que, utilizando lua, definan comportamiento específico para cualquier modelo físico y gráfico que se quiera incluir. Lo que haría posible el desarrollo de misiones en las que poder enfrentarse a rivales o enemigos controlados por el propio programa. Para ello habría que construir una máquina de estados relacionada con los scripts de lua.

15.2 CONFIGURACIÓN VISUAL DEL HUD

Tal y como ha quedado la configuración de los elementos visibles en la clase HelicopterGUI, pretender recolocar cualquiera de ellos en otro punto de la pantalla conllevaría un tedioso trabajo de prueba y error modificando el fichero **GUI.cfg,** con un elevado riesgo de equivocación que provocaría un funcionamiento no esperado o incluso la desestimación del fichero a la hora de cargarlo.

Por ello proponemos dos posibles mejoras respecto a este tema:

Primero: Refinar el método **rellenaVisibles** de la clase **HelicopterGUI** acercándolo a la implementación de un analizador de lenguaje.

Segundo: Implementar un método para configurar gráficamente el HUD, éste método debería recoger la posición de los elementos en la pantalla y sobrescribir el archivo **GUI.cfg.**

15.3 CONTROL AUTOMÁTICO

Respecto al control automático del helicóptero, una posible mejora sería susituir el controlador Matlab rudimentario implementado como prueba de sockets por el proyecto desarrollado este mismo curso por Mario Chueca Burgueño, Javier López Carreras e Iván Rebollo Martínez titulado *"Modelización de un sistema no lineal de vuelo"* y supervisado por José Manuel de la Cruz García.

15.4 MODELO FÍSICO

Al modelo físico todavía le faltan muchas cosas por implementar para que resulte completamente real. Por ejemplo, ahora mismo el helicóptero resiste los impactos, la inmersión en el agua y todo tipo de eventos que a un modelo real afectarían de manera grave. La velocidad de giro de las palas no siempre coincide con la que tendría un modelo real.

En definitiva, se propone el autentificar el comportamiento del helicóptero para acercarlo aún más al comportamiento real.

15.5 MULTIJUGADOR

Una vez implementado el paso de mensajes mediante Sockets, parece obvio implementar un sistema que gestione la presencia de varios jugadores en un mismo escenario. Si se ha desarrollado un sistema de misiones se podrían diseñar misiones de competencia, colaboración para alcanzar un mismo objetivo, etc, etc ...

16. BIBLIOGRAFÍA

Páginas en Internet:

[1] http://serdis.dis.ulpgc.es/~itis-iga/PracOGL2001/
Contiene diversas prácticas realizadas en OpenGL. Una de ellas es un simulador de un helicóptero.

[2] http://www.alphamacsoftware.com/
Enlace a un ejecutable de un simulador de helicópteros de radio control para Macintosh Mac OS X.

[3] http://autopilot.sourceforge.net/sim.html
Simulador de un helicóptero de radio control hecho en OpenGL con libreríasmatemáticas y físicas de U.S. Naval Research Laboratory, realizadas por Aaron Kahn, Suresh Kannan y Dr. Eric Jonson.

[4] http://www.mundodirectx.galeon.com
Página que tiene documentación y el software de desarrollo para utilizar DirectX8.

[5] http://www.spheregames.com/articles.asp
Diversos artículos sobre técnicas de dibujo en 3D.

[6] http://www.gamedev.net
Gran variedad de artículos sobre el desarrollo de juegos, tutoriales, foros y demás recursos.

[7] http://www.robot-frog.com/3d/index.html
Tutorial sobre la implementación de terrenos en 3D.

[8] http://www.codesampler.com/dx8src.htm
Código de ejemplos de juegos implementados con DirectX.

[9] http://www.deus.co.uk/3dgraphics.shtml
Página con diversos tutoriales sobre gráficos en 3D.

[10] http://msdn.microsoft.com
Página de ayuda de Microsoft. Contiene entre otras cosas toda la ayuda de los métodos utilizados por DirectX.

[11] http://www.mathworks.com/access/helpdesk/help/helpdesk.html
Página de ayuda para usar Matlab y Simulink. Contiene ayuda sobre todos los métodos utilizados en la creación de S-Functions.

[12] http://cs.ecs.baylor.edu/~donahoo/practical/CSockets/practical/
Información sobre cómo implementer sockets usando Winsock.

[13] http://www.cica.es/formacion/JavaTut/cap9/socket.html
Tutorial sobre sockets en Java.

[14] http://www.fmod.org/
Todo lo relacionado con la librería fmod: tutoriales, ejemplos, etc.

[15] *http://www.xbdev.net/3dformats/x/xfileformat.php*
Página con información sobre el formato de ficheros de DirectX

[16] http://www.xbdev.net/3dformats/x/file_specs/DX113Spec.doc
Enlace al documento oficial de Microsoft con la especificación de los ficheros .x

[17] http://www.andypike.com/tutorials/directx8/
Contiene un práctico tutorial sobre el uso de DirectX8.

[18] http://www.engin.umd.umich.edu/CIS/course.des/cis487/

[19] http://www.geocities.com/SiliconValley/2151/drawprim.html
Contiene un ejemplo del uso de la función DrawPrimitive de DirectX.

[20] http://www.codeproject.com/
Gran variedad de artículos sobre el desarrollo de juegos, tutoriales, foros y demás recursos.

[21] http://www.thecodeproject.com/
Gran variedad de artículos sobre el desarrollo de juegos, tutoriales, foros y demás recursos.

[22] http://www.mvps.org/directx/articles/d3dxmesh.htm - meshcreate
Artículo donde se explica la creación y pintado de Mallas en DirectX.

[23] http://www.yindo.com/lua/Lua_reference_manual_4_0.html
Manual de referencia de Lua.

[24] http://www.jucs.org/jucs_10_7/luainterface_scripting_the_.net
Artículo sobre cómo desarrollar el interfaz Lua – C.

[25] http://www.lua.org/
Página principal del lenguaje de programación Lua.

Libros:

[1] Kelly Dempski **"**Real-time rendering tricks and techniques in DirectX"
[2] Mason McCuskey "Special effects game programming with DirectX"
[3] Alex Varanese "Game scripting mastery"
[4] Eric Lengyel "Mathematics for 3d game programming and computer graphics"
[5] David M. Bourg "Physics for game developers"
[6] Greg Snook "Real-Time 3d terrain engines using c++ and DirectX 9"
[7] Peter Walsh "Advanced 3d game programming with DirectX 9.0"
[8] Frank D. Luna "Introduction to 3d game programming with DirectX 9.0"
[9] William H. Press…[et Al.] "Numerical recipes in c++: the art of scientific computing"
[10] Roberto Ierusalimschy "Programming in Lua"
[11] Tom Gutschmidt "Game programming with Pytho, Lua and Ruby"

[12] "Lua reference Manual"
[13] "DirectX 3D Graphics Programming Bible"
[14] F.S. Hill "Computer graphics using OpenGL"
[15] Guillermo Martínez Sánchez "Modelación del sistema no lineal de un helicóptero" Trabajo de investigación del curso 2002/2003

17. PALABRAS CLAVE

Helicóptero
Simulador
Terreno Infinito
Colisiones
DirectX
Sockets
Dial
FMOD
Lua
3D

18. AUTORIZACIÓN

Los abajo firmantes autorizan a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Xavier Paolo Burgos Artizzu

Ángel Luis Diez Hernández

Ángel Iglesias Sánchez

Madrid, a 29 de Junio de 2005